Department of
Computer Science
University of Illinois
at Urbana-Champaign

Technical Report

SIPL '95

Second ACM SIGPLAN Workshop on State in Programming Languages

San Francisco, California
January 22, 1995

UIUCDCS-R-95-1900

UILU-ENG-95-1702

Seco kshop on St guages

UIUCDCS-R-95-1900

UILU-ENG-95-1702

SIPL '95

Second ACM SIGPLAN Workshop on State in Programming Languages

San Francisco, California
January 22, 1995

UIUCDCS-R-95-1900 UILU-ENG-95-1702

Foreword

Programming languages have been state-based since their inception. After a period of relative unpopularity, when research focused on declarative languages, interest in the treatment of state has been renewed. Research is increasingly devoted to finding a symbiotic relationship between the semantic foundations of declarative languages and the pragmatic handling of state in more conventional languages.

The ACM SIGPLAN Workshop on *State in Programming Languages* brings together researchers from various areas interested in the common issues of state manipulation in high-level programming languages. The first workshop in the series (SIPL '93) was held in Copenhagen in June, 1993. This proceedings contains the papers presented at the second workshop (SIPL '95) held on January 22, 1995 in San Francisco, California.

We received ten submissions in response to the electronic call for papers, all of which were of very high quality. The program committee recommended seven submissions for presentation at the workshop. All these papers appear in this proceedings.

I would like to thank the members of the Program Committee for their timely reviews within a short time frame. Special thanks go to my co-organizer, Peter O'Hearn, and to Bonnie Howard for assistance at University of Illinois. We thank ACM SIGPLAN for agreeing to sponsor this event and to Ron Cytron and his crew for the joint organization with POPL. Finally, thanks to Jonathan Springer for putting SIPL on the world wide web. Please watch this web page for future announcements related to SIPL: "`http://vesuvius.cs.uiuc.edu:8080/sipl/index.html`".

Uday S. Reddy
SIPL Program Chair

Program Committee

Foreword

Contents

Session 1:

Session 2:

Session 3:

Session 4:

Formalizing Hoare Logic *

Joshua E. Caplan
Department of Computer Science
University of Illinois at Urbana-Champaign
caplan@cs.uiuc.edu

December 22, 1994

Abstract

We present a new formalization of Hoare Logic (using the Edinburgh Logical Framework [5]) and examine the issues raised. Our approach improves upon current LF representations of Hoare Logic and, via the LF type theory, generates for free a logic for a fragment of Idealized Algol [13], including proofs about procedures.

1 Introduction

In this paper, *formalizing* a logic L means defining a signature Σ_L in the language of some meta-logic, like the Edinburgh Logical Framework (LF) [5], such that Σ_L adequately represents L. L is called the *object logic*, and Σ_L its *representation* or *encoding*. There are (at least) three benefits to formalization of a logic:

1. Philosophical: The search for an adequate encoding usually brings to light important properties of the object logic.
2. Theoretical: Through the expressive power of the meta-logic, we obtain from the encoding a non-trivial, yet decidable and conservative extension of the object logic.
3. Practical: We can plug the encoding into any existing implementation of the meta-logic to immediately obtain a proof checker and/or theorem prover for the object logic.

We will describe a new formalization of Hoare Logic in LF which improves on those which exist in the literature, and in so doing illustrate all of the above benefits.

The first will surface when we elucidate the property of Hoare Logic which inhibits a straightforward encoding, motivating the creation of new syntax and two new calculi which repair this defect without fundamentally altering the notion of provability.

The second will appear when we then formalize one of the new calculi and obtain, by virtue of the LF type theory, a logic which allows us to reason about proper procedures as well as simple commands, and to reuse the proofs for their calls, *as long as the active arguments don't interfere with the procedures.* This proviso is the basis of Syntactic Control of Interference [9, 10, 14], so it is not surprising that our system has some features which seem to correspond to aspects of the typing system of P. W. O'Hearn and R. D. Tennent [9], although we arrived at our techniques independently of their work.

We have exploited benefit number three to mechanically verify the examples in section 5.

*This work partially supported by NASA grant NAG 1-613 (I-CLASS)

1.1 Related Work

There are two published formalizations of Hoare Logic in the LF, which appear in [3, 7]. Hoare Logic is not easy to formalize because weakening and contraction are admissible rules in the LF, which can lead to certain invalid formulas of the naive encoding being inhabited (see section 2.1).

In the first formalization, new judgment forms denoting non-interference and non-identity of program variables are used to express a correct assignment axiom. This approach is reminiscent of J. C. Reynolds' Specification Logic [11, 13], but as a result seems not to be an entirely faithful representation of Hoare Logic. For example, having obtained a proof term t in a context $\Gamma = \{x, y : \mathbf{var}, z : x \sharp_{\mathbf{var}} y, z^* : y \sharp_{\mathbf{var}} x\}$ as prescribed by I. A. Mason in [7], contraction yields $t[x/y]$ in context $\Gamma' = \{x : \mathbf{var}, z, z^* : x \sharp_{\mathbf{var}} x\}$ which may correspond to some non-existent proof in Hoare Logic (based as it is on flawed assumptions).

In the second formalization, program variables used in Hoare triples are bound by the LF λ. All rules are given relative to these bound variables, so in order to have a finitary LF signature, the number and order of those variables must be fixed and finite. Thus the formalization is a *family* of signatures, one for each choice of variable set, and each one turns out to be adequate for a fragment of our calculus QHL_0 (section 2.3), but certainly not for full Hoare Logic.

The signature we give neither employs auxiliary judgments about non-interference nor fixes the set of program variables in advance. We avoid these pitfalls, and admit far greater extensibility, by unabashedly modifying the object logic before encoding it, incorporating a disguised form of affine contexts. Such contexts for program variables have appeared in earlier work, e.g. [9], and other researchers have found it natural to reformulate a program logic before formalizing it [6].

2 Repairing Hoare Logic

2.1 Hoare Logic

Recall the usual natural-deduction formulation of Hoare Logic (Table 1), defined relative to an underlying first-order language L and first-order L-theory of expressions T. There is only one judgment form, asserting the validity of a Hoare triple. The other judgment form which appears in the rule of consequence (Con) is that of T, so we consider Con to be schematic not only in the meta-variables P', P, Q, Q', c but also in the derivations of the L-formulas $P' \Rightarrow P$ and $Q \Rightarrow Q'$. As presented, there are no "closed" Hoare triples, because every program must contain at least one assignment (with a free identifier).

Notice that free identifiers in first-order formulas are implicitly universally quantified; a formula's validity is immune to substitution because universal elimination is a sound rule in first-order logic. Free identifiers in Hoare triples do not enjoy the same genericity. For example, the triple $\phi \stackrel{\text{dfn}}{=} \{\mathbf{true}\}\, x := y + 1\, \{y < x\}$ is valid, but $\phi[z/x, z/y] = \{\mathbf{true}\}\, z := z + 1\, \{z < z\}$ is not. *Validity is not preserved by substitution in Hoare Logic.*

In the simple programming language of Hoare Logic, substitution never arises, and all distinct identifiers implicitly denote *non-interfering* program variables; that is, assigning to one will not affect the value of the other. In a language with procedures, such as Algol, the assignment axiom breaks down because substitutions occur with each procedure call. The higher-order nature of the LF also permits substitution, but the inhabitability of any type in the LF is always preserved; thus we cannot directly encode the form of Hoare Logic given in Table 1.

Instead, we give an essentially equivalent formulation of Hoare Logic which is amenable to formalization. LF substitution then includes the copy rule of Algol, so our resulting system handles procedures, too.

(Assign) $\dfrac{}{\{P[t/x]\}\, x := t\, \{P\}}$ (Seq) $\dfrac{\{P\}\, c\, \{Q\} \quad \{Q\}\, d\, \{R\}}{\{P\}\, c\, ;\, d\, \{R\}}$

(If) $\dfrac{\{P \wedge b\}\, c\, \{Q\} \quad \{P \wedge \neg b\}\, d\, \{Q\}}{\{P\}\, \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ d\, \{Q\}}$ (While) $\dfrac{\{P \wedge b\}\, c\, \{P\}}{\{P\}\, \texttt{while}\ b\ \texttt{do}\ c\, \{P \wedge \neg b\}}$

(Con) $\dfrac{P' \Rightarrow P \quad \{P\}\, c\, \{Q\} \quad Q \Rightarrow Q'}{\{P'\}\, c\, \{Q'\}}$

Table 1: Hoare Logic.

2.2 Quantified Hoare Logic: Syntax

Definition 2.1 Let L be a first-order language, V the set of variables, E the set of L-terms, F the set of L-formulas, and Q the set of quantifier-free L-formulas. The set of formulas ⟨spec⟩ of Quantified Hoare Logic (over L) is given by the following grammar:

$$\begin{array}{lll} \langle \text{command} \rangle & ::= & V := E \mid (\langle \text{command} \rangle\ ;\ \langle \text{command} \rangle) \mid \\ & & (\texttt{if}\ Q\ \texttt{then}\ \langle \text{command} \rangle\ \texttt{else}\ \langle \text{command} \rangle) \mid (\texttt{while}\ Q\ \texttt{do}\ \langle \text{command} \rangle) \\ \langle \text{spec} \rangle & ::= & \{F\}\, \langle \text{command} \rangle\, \{F\} \mid \forall V\, \langle \text{spec} \rangle \end{array}$$

We call the formulas of Quantified Hoare Logic *specs*. It is obvious, but nevertheless important, that every spec consists of zero or more ∀-binding identifier occurrences (the *prefix*) followed by a Hoare triple (the *matrix*).

Definition 2.2 Substitution is defined for Quantified Hoare Logic just as for Hoare Logic, except that ∀ is a binding operator.

2.3 The Calculus QHL_0

The most natural proof calculus for Quantified Hoare Logic, called QHL_0, is given by the axioms and inference rules of Table 2. We say $\vdash_0 \phi$ just in case ϕ is derivable via the rules.

In this calculus there are no "active open" derivable specs, i.e., the active identifiers are all bound by ∀ (Lemma 2.4). Observe that the quantifier ∀ does not act as a medium for instantiation, as does the universal quantifier of first-order logic. There is no analogous ∀-elimination rule in QHL_0. Rather, through the assignment axiom, we force identifiers appearing on the left-hand sides of assignments in the command portion of derivable specs to be bound by ∀. The prefix acts as a context keeping track of actively-occurring identifiers (and protecting them from substitutions). Within any QHL_0 derivation, every triple will have an identical prefix.

Having changed the language and proof system of Hoare Logic, it is desirable to know that we have not fundamentally altered the notion of provability.

Definition 2.3 An identifier which appears on the left-hand side of an assignment in a spec is called *active* in that spec.

Lemma 2.4 *Specs derivable in* QHL_0 *have no free active identifiers.*

Proof. By an easy induction on derivations. □

$$(\text{Assign}_{n,m}) \quad \frac{}{\forall \vec{x}\,\{P[t/x_m]\}\, x_m := t\,\{P\}} \qquad (\text{Seq}_n) \quad \frac{\forall \vec{x}\,\{P\}\, c\,\{Q\} \quad \forall \vec{x}\,\{Q\}\, d\,\{R\}}{\forall \vec{x}\,\{P\}\, c\,;\, d\,\{R\}}$$

$$(\text{If}_n) \quad \frac{\forall \vec{x}\,\{P \wedge b\}\, c\,\{Q\} \quad \forall \vec{x}\,\{P \wedge \neg b\}\, d\,\{Q\}}{\forall \vec{x}\,\{P\}\, \texttt{if } b \texttt{ then } c \texttt{ else } d\,\{Q\}} \qquad (\text{While}_n) \quad \frac{\forall \vec{x}\,\{P \wedge b\}\, c\,\{P\}}{\forall \vec{x}\,\{P\}\, \texttt{while } b \texttt{ do } c\,\{P \wedge \neg b\}}$$

$$(\text{Con}_n) \quad \frac{P' \Rightarrow P \quad \forall \vec{x}\,\{P\}\, c\,\{Q\} \quad Q \Rightarrow Q'}{\forall \vec{x}\,\{P'\}\, c\,\{Q'\}}$$

$m, n \in \omega, 0 \leq n, 1 \leq m \leq n, \vec{x} \equiv x_1 \ldots x_n$ in all rules

Table 2: Calculus QHL_0 for Quantified Hoare Logic.

Proposition 2.5 *Let ϕ be a spec, $\hat{\phi}$ its matrix, X the set of identifiers bound in its prefix, and A the set of its active identifiers. Then*

$$(\vdash_0 \phi) \Longleftrightarrow (\vdash_{\text{HL}} \hat{\phi} \wedge A \subseteq X).$$

Proof. ($\Longleftarrow$) Remove the prefix from every spec in a QHL_0 derivation of ϕ, and apply Lemma 2.4; ($\Longrightarrow$) add the prefix to every triple in a Hoare Logic derivation of $\hat{\phi}$. □

Theorem 2.6 *The provable formulas of QHL_0 are closed under substitution, i.e., for any formula ϕ of Quantified Hoare Logic*

$$\vdash_0 \phi \Longrightarrow \vdash_0 \phi[\vec{t}/\vec{x}].$$

Proof. Without loss of generality, we show that for any Hoare triple ϕ, collection of distinct identifiers $\vec{x}, y$, and term s not containing $\vec{x}$ free,

$$\vdash_0 \forall \vec{x}\phi \Longrightarrow \vdash_0 \forall \vec{x}\phi[s/y].$$

The proof proceeds by induction on derivations:

Base Case $\phi \equiv \{P[t/x_m]\}\, x_m := t\,\{P\}$. Then

$$\begin{aligned}(\forall \vec{x}\phi)[s/y] &= \forall \vec{x}\,\{P[t/x_m][s/y]\}\, x_m := t[s/y]\,\{P[s/y]\} \\ &= \forall \vec{x}\,\{P[s/y][t[s/y]/x_m]\}\, x_m := t[s/y]\,\{P[s/y]\},\end{aligned}$$

which is derivable by the assignment axiom.

Induction Step Follows routinely from the definition of substitution. □

Unfortunately, this calculus does not lend itself to formalization. In the LF, an inference rule is represented as a constant whose type, schematic in the meta-variables of the rule, is the type of maps from proofs of the antecedent judgments to proofs of the consequent judgment. But the type system of LF cannot express the rules of QHL_0 as types schematic in n and m, because the physical shape of the formulas in the antecedent and consequent vary in those meta-variables. Therefore each rule in Table 2 would require infinitely-many constants in an LF signature, one for each n and m. Because infinite-length signatures are not permitted, we must develop a more homogeneous calculus.

$$(\forall\text{Intro})^* \quad \frac{\phi}{\forall x \phi} \qquad (\text{Exch}) \quad \frac{\forall x \forall y \phi}{\forall y \forall x \phi}$$

$$(\forall\text{IntroUn})^* \quad \frac{\begin{array}{c}[\psi] \\ \phi \quad \forall x \psi\end{array}}{\forall x \phi} \qquad (\forall\text{IntroBin})^* \quad \frac{\begin{array}{c}[\psi, \theta] \\ \phi \quad \forall x \psi \quad \forall x \theta\end{array}}{\forall x \phi}$$

$$(\text{Assign}) \quad \frac{}{\forall x \, \{P[t/x]\} \, x := t \, \{P\}} \qquad (\text{Seq}) \quad \frac{\{P\} \, c \, \{Q\} \quad \{Q\} \, d \, \{R\}}{\{P\} \, c \, ; \, d \, \{R\}}$$

$$(\text{If}) \quad \frac{\{P \wedge b\} \, c \, \{Q\} \quad \{P \wedge \neg b\} \, d \, \{Q\}}{\{P\} \, \texttt{if } b \texttt{ then } c \texttt{ else } d \, \{Q\}} \qquad (\text{While}) \quad \frac{\{P \wedge b\} \, c \, \{P\}}{\{P\} \, \texttt{while } b \texttt{ do } c \, \{P \wedge \neg b\}}$$

$$(\text{Con}) \quad \frac{P' \Rightarrow P \quad \{P\} \, c \, \{Q\} \quad Q \Rightarrow Q'}{\{P'\} \, c \, \{Q'\}}$$

*Provided x is free only in ϕ and/or the discharged hypothesis occurrences.

Table 3: Calculus QHL_1 for Quantified Hoare Logic.

2.4 The Calculus QHL_1

Such a calculus is given in Table 3. We denote derivability in QHL_1 by $\vdash_1$. The rules are clearly divided into *structural* rules, including the ∀Intro rules and Exch, and the *compositional* rules, comprising the Hoare Logic analogues. The proof-theoretic equivalence of QHL_0 and QHL_1 is the main theorem of this section.

Theorem 2.7 *For any spec ϕ, $\vdash_0 \phi \iff \vdash_1 \phi$.*

Proof. To prove the $\Longrightarrow$ direction, we show that the rules of QHL_0 are all derivable in QHL_1, inductively applying structural rules to add bindings to the compositional rules. In the case of $\text{Assign}_{n,m}$ we must use the Exch rule. For example,

$$\frac{\dfrac{[\{P\} \, c \, \{Q\}] \quad [\{Q\} \, d \, \{R\}]}{\{P\} \, c \, ; \, d \, \{R\}}(\text{Seq}) \quad \forall x \, \{P\} \, c \, \{Q\} \quad \forall x \, \{Q\} \, d \, \{R\}}{\forall x \, \{P\} \, c \, ; \, d \, \{R\}}(\forall\text{IntroBin})$$

represents Seq_1.

To prove the $\Longleftarrow$ direction, we observe that the compositional rules of QHL_1 are all special cases of their counterparts in QHL_0, and that the structural rules are all admissible (though not derivable) rules of QHL_0. □

Thus the provable specs in QHL_1 are just the provable triples of Hoare Logic with (at least) their active identifiers bound by ∀. Intuitively, the reason QHL_1 is equivalent to QHL_0 is that there are only three kinds of rules in Hoare Logic: axioms (0 antecedents), unary rules (1 antecedent), and binary rules (2 antecedents). Thus we only need two ∀Intro rules to generate the inference rules of QHL_0 from plain Hoare Logic, and a third ∀Intro rule and Exch to get the assignment axioms.

Corollary 2.8 *The provable formulas of* QHL_1 *are closed under substitution.*

In this calculus, the contextual role of $\forall$ is transparent. Notice that we have weakening (rule $\forall$Intro) and exchange (rule Exch and its many derived forms), but no contraction or elimination. Evidently from the example in section 2.1, contraction lies at the root of the problem with substitution in plain Hoare Logic. When substitution becomes formally available in the next section, the omission of elimination rules for $\forall$ from the calculus will be essential in obtaining semantic adequacy.

3 The Formalization

We define our signature assuming encodings Σ_{FOL} of first-order logic and Σ_T of a finitely-axiomatizable first-order theory T.

3.1 Syntax

We assume the type constants **exp** for terms of T and **assert** for formulas of T are defined in Σ_{FOL}. Within Σ_{FOL} are purely logical constructors, including

$$\begin{aligned} = &: \mathbf{exp} \to \mathbf{exp} \to \mathbf{assert} \\ \neg &: \mathbf{assert} \to \mathbf{assert} \\ \Rightarrow, \wedge, \vee &: \mathbf{assert} \to \mathbf{assert} \to \mathbf{assert} \\ \forall, \exists &: (\mathbf{exp} \to \mathbf{assert}) \to \mathbf{assert} \end{aligned}$$

and Σ_T contains constructors of the form

$$\begin{aligned} c &: \mathbf{exp} \\ f &: \overbrace{\mathbf{exp} \to \ldots \to \mathbf{exp}}^{n} \to \mathbf{exp} \\ r &: \overbrace{\mathbf{exp} \to \ldots \to \mathbf{exp}}^{m} \to \mathbf{assert} \end{aligned}$$

for each non-logical constant c, n-ary function symbol f, and m-ary predicate symbol r of T.

The syntactic categories of Quantified Hoare Logic are easily declared:

$$\mathbf{var}, \mathbf{comm}, \mathbf{spec} : \mathbf{TYPE}$$

We follow Mason [7] in declaring a separate type **var** for program variables and giving no constructors for it, but only a coercion

$$\uparrow : \mathbf{var} \to \mathbf{exp},$$

forcing the only normal forms of type **var** (in ground contexts) to be LF identifiers. The syntactic judgment **QF** will be discussed in the next subsection.

Command and spec constructors look like

$$\begin{aligned} := &: \mathbf{var} \to \mathbf{exp} \to \mathbf{comm} \\ ; &: \mathbf{comm} \to \mathbf{comm} \to \mathbf{comm} \\ \mathtt{if} &: \textstyle\prod_{b:\mathbf{assert}} \mathbf{QF}(b) \to \mathbf{comm} \to \mathbf{comm} \to \mathbf{comm} \\ \mathtt{while} &: \textstyle\prod_{b:\mathbf{assert}} \mathbf{QF}(b) \to \mathbf{comm} \to \mathbf{comm} \\ \{\_\}\_\{\_\} &: \mathbf{assert} \to \mathbf{comm} \to \mathbf{assert} \to \mathbf{spec} \\ \forall &: (\mathbf{var} \to \mathbf{spec}) \to \mathbf{spec} \end{aligned}$$

We will freely use infix for operators like $=$ and $:=$, postfix for the $\uparrow$ operator, and the abbreviation $\forall x\ \phi$ for $\forall(\lambda x\,.\,\phi)$ in what follows.

3.2 Judgment Forms

There are three: validity of an assertion in the underlying first-order theory **T**, validity of a partial correctness formula **H**, and a syntactic judgment **QF** first seen in [7] which indicates that an assertion is quantifier-free and can be used as the conditional expression in an `if` or `while` command.

$$\begin{aligned} \mathbf{T}, \mathbf{QF} &: \mathbf{assert} \to \mathbf{TYPE} \\ \mathbf{H} &: \mathbf{spec} \to \mathbf{TYPE} \end{aligned}$$

3.3 Axioms and Inference Rules

The rules of QHL_1, translated into the formalism of the LF, and the rules concerning **QF** are shown in Table 4.

3.4 Adequacy

Call the signature obtained from combining Σ_{FOL}, Σ_T, and the preceding declarations Σ^T_{QHL}.

Theorem 3.1 (Mason [7]) *The signature Σ^T_{QHL} is syntactically adequate for Quantified Hoare Logic with respect to context*

$$\Gamma \stackrel{\mathrm{dfn}}{=} [x_1 : \mathbf{var}, \ldots, x_n : \mathbf{var}],$$

i.e., for each type $\theta \equiv \mathbf{var}, \mathbf{exp}, \mathbf{assert}, \mathbf{comm}, \mathbf{spec}$ there is a compositional bijection τ_θ between well-formed LF $\beta\eta$-normal forms of type θ and expressions of the corresponding syntactic category (with free identifiers among those in Γ) in Quantified Hoare Logic.

Theorem 3.2 *The signature Σ^T_{QHL} is semantically adequate for QHL_1 with respect to the empty context, i.e., for every spec ϕ there exists a compositional bijection τ_ϕ between well-formed LF $\beta\eta$-normal forms of type $\mathbf{H}(\phi)$ and derivations of ϕ in QHL_1.*

Proof. An uninteresting pair of inductions on normal forms and expressions. □

We abbreviate argument lists $x_1\uparrow\ x_2\uparrow\ \ldots\ x_n\uparrow$ to $\vec{x}\uparrow$. We have the following as a corollary to the adequacy theorems and the proof of Theorem 2.7:

Corollary 3.3 *The following are derived inference rules, for each $1 \le m \le n \in \omega$:*

$$\begin{aligned} \mathrm{AsgnAx}_{n,m} \;:\; & \textstyle\prod_{P:\overbrace{\mathbf{exp}\to\ldots\to\mathbf{exp}}^{n}\to\mathbf{assert}} \prod_{t:\overbrace{\mathbf{var}\to\ldots\to\mathbf{var}}^{n}\to\mathbf{exp}} \\ & \mathbf{H}(\forall x_1 \ldots \forall x_n \{P\, x_1\uparrow\ \ldots\ x_{m-1}\uparrow\ (t\,\vec{x})\, x_{m+1}\uparrow\ \ldots\ x_n\uparrow\}\, x_m := (t\,\vec{x})\,\{P\,\vec{x}\uparrow\}) \\ \mathrm{SeqRule}_n \;:\; & \textstyle\prod_{P,Q,R:\overbrace{\mathbf{var}\to\ldots\to\mathbf{var}}^{n}\to\mathbf{assert}} \prod_{c,d:\overbrace{\mathbf{var}\to\ldots\to\mathbf{var}}^{n}\to\mathbf{comm}} \\ & \mathbf{H}(\forall x_1 \ldots \forall x_n \{P\,\vec{x}\}\, c\,\vec{x}\,\{Q\,\vec{x}\}) \to \\ & \mathbf{H}(\forall x_1 \ldots \forall x_n \{Q\,\vec{x}\}\, d\,\vec{x}\,\{R\,\vec{x}\}) \to \\ & \mathbf{H}(\forall x_1 \ldots \forall x_n \{P\,\vec{x}\}\, c\,\vec{x}\ ;\ d\,\vec{x}\,\{R\,\vec{x}\}) \end{aligned}$$

$$
\begin{array}{rcl}
\mathrm{QF}= & : & \prod_{t,u:\mathbf{exp}} \mathbf{QF}(t = u) \\
\mathrm{QF}r & : & \prod_{t_1,\ldots,t_m:\mathbf{exp}} \mathbf{QF}(r\, t_1 \ldots t_m) \qquad [r \text{ an } m\text{-ary pred. sym. from } T] \\
\mathrm{QF}\neg & : & \prod_{b:\mathbf{assert}} \mathbf{QF}(b) \to \mathbf{QF}(\neg b) \\
\mathrm{QF}\chi & : & \prod_{a,b:\mathbf{assert}} \mathbf{QF}(a) \to \mathbf{QF}(b) \to \mathbf{QF}(a\, \chi\, b) \qquad [\chi \equiv \wedge, \vee, \Rightarrow] \\
\mathrm{AsgnAx}_{1,1} & : & \prod_{P:\mathbf{exp}\to\mathbf{assert}} \prod_{t:\mathbf{var}\to\mathbf{exp}} \\
 & & \quad \mathbf{H}(\forall(\lambda x\,.\,\{P\,(t\,x)\}\, x := t\, x\, \{P\, x\uparrow\})) \\
\mathrm{SeqRule}_0 & : & \prod_{P,Q,R:\mathbf{assert}} \prod_{c,d:\mathbf{comm}} \\
 & & \quad \mathbf{H}(\{P\}\, c\, \{Q\}) \to \mathbf{H}(\{Q\}\, d\, \{R\}) \to \mathbf{H}(\{P\}\, c\, ;\, d\, \{R\}) \\
\mathrm{IfRule}_0 & : & \prod_{P,b,Q:\mathbf{assert}} \prod_{c,d:\mathbf{comm}} \prod_{s:\mathbf{QF}(b)} \\
 & & \quad \mathbf{H}(\{P \wedge b\}\, c\, \{Q\}) \to \mathbf{H}(\{P \wedge \neg b\}\, d\, \{Q\}) \to \mathbf{H}(\{P\}\, \mathtt{if}\, b\, s\, c\, d\, \{Q\}) \\
\mathrm{WhileRule}_0 & : & \prod_{P,b:\mathbf{assert}} \prod_{c:\mathbf{comm}} \prod_{s:\mathbf{QF}(b)} \\
 & & \quad \mathbf{H}(\{P \wedge b\}\, c\, \{P\}) \to \mathbf{H}(\{P\}\, \mathtt{while}\, b\, s\, c\, \{P \wedge \neg b\}) \\
\mathrm{ConRule}_0 & : & \prod_{P',P,Q,Q':\mathbf{assert}} \prod_{c:\mathbf{comm}} \mathbf{T}(P' \Rightarrow P) \to \mathbf{T}(Q \Rightarrow Q') \to \\
 & & \quad \mathbf{H}(\{P\}\, c\, \{Q\}) \to \mathbf{H}(\{P'\}\, c\, \{Q'\}) \\
\mathrm{Exch} & : & \prod_{f:\mathbf{var}\to\mathbf{var}\to\mathbf{spec}} \\
 & & \quad \mathbf{H}(\forall(\lambda x\,.\,\forall(\lambda y\,.\,f\, x\, y))) \to \mathbf{H}(\forall(\lambda y\,.\,\forall(\lambda x\,.\,f\, x\, y))) \\
\forall\mathrm{IntroAx} & : & \prod_{f:\mathbf{var}\to\mathbf{spec}} \\
 & & \quad (\prod_{x:\mathbf{var}} \mathbf{H}(f\, x)) \to \mathbf{H}(\forall\, f) \\
\forall\mathrm{IntroUn} & : & \prod_{f,g:\mathbf{var}\to\mathbf{spec}} \\
 & & \quad \Big(\prod_{x:\mathbf{var}} \mathbf{H}(f\, x) \to \mathbf{H}(g\, x)\Big) \to \Big(\mathbf{H}(\forall\, f) \to \mathbf{H}(\forall\, g)\Big) \\
\forall\mathrm{IntroBin} & : & \prod_{f,g,h:\mathbf{var}\to\mathbf{spec}} \\
 & & \quad \Big(\prod_{x:\mathbf{var}} \mathbf{H}(f\, x) \to \mathbf{H}(g\, x) \to \mathbf{H}(h\, x)\Big) \to \Big(\mathbf{H}(\forall\, f) \to \mathbf{H}(\forall\, g) \to \mathbf{H}(\forall\, h)\Big)
\end{array}
$$

Table 4: Axioms and Rules of Σ^T_{QHL}.

$$
\begin{aligned}
\text{IfRule}_n \;:\; & \textstyle\prod_{P,b,Q:\overbrace{\mathbf{var}\to\ldots\to\mathbf{var}}^{n}\to\mathbf{assert}} \prod_{c,d:\overbrace{\mathbf{var}\to\ldots\to\mathbf{var}}^{n}\to\mathbf{comm}} \prod_{s:(\Pi_{\vec{y}:\mathbf{var}}\mathbf{QF}(b\,\vec{y}))} \\
& \mathbf{H}(\forall x_1\ldots\forall x_n\,\{P\,\vec{x}\wedge b\,\vec{x}\}\,c\,\vec{x}\,\{Q\,\vec{x}\}) \to \\
& \mathbf{H}(\forall x_1\ldots\forall x_n\,\{P\,\vec{x}\wedge\neg(b\,\vec{x})\}\,d\,\vec{x}\,\{Q\,\vec{x}\}) \to \\
& \mathbf{H}(\forall x_1\ldots\forall x_n\,\{P\,\vec{x}\}\,\mathtt{if}\,(b\,\vec{x})\,(s\,\vec{x})\,(c\,\vec{x})\,(d\,\vec{x})\,\{Q\,\vec{x}\}) \\
\text{WhileRule}_n \;:\; & \textstyle\prod_{P,b:\overbrace{\mathbf{var}\to\ldots\to\mathbf{var}}^{n}\to\mathbf{assert}} \prod_{c:\overbrace{\mathbf{var}\to\ldots\to\mathbf{var}}^{n}\to\mathbf{comm}} \prod_{s:(\Pi_{\vec{y}:\mathbf{var}}\mathbf{QF}(b\,\vec{y}))} \\
& \mathbf{H}(\forall x_1\ldots\forall x_n\,\{P\,\vec{x}\wedge b\,\vec{x}\}\,c\,\vec{x}\,\{P\,\vec{x}\}) \to \\
& \mathbf{H}(\forall x_1\ldots\forall x_n\,\{P\,\vec{x}\}\,\mathtt{while}\,(b\,\vec{x})\,(s\,\vec{x})\,(c\,\vec{x})\,\{P\,\vec{x}\wedge\neg(b\,\vec{x})\}) \\
\text{ConRule}_n \;:\; & \textstyle\prod_{P',P,Q,Q':\overbrace{\mathbf{var}\to\ldots\to\mathbf{var}}^{n}\to\mathbf{assert}} \prod_{c:\overbrace{\mathbf{var}\to\ldots\to\mathbf{var}}^{n}\to\mathbf{comm}} \\
& (\textstyle\prod_{\vec{y}:\mathbf{var}}\mathbf{T}(P'\,\vec{y}\Rightarrow P\,\vec{y})) \to (\prod_{\vec{y}:\mathbf{var}}\mathbf{T}(Q\,\vec{y}\Rightarrow Q'\,\vec{y})) \to \\
& \mathbf{H}(\forall x_1\ldots\forall x_n\,\{P\,\vec{x}\}\,c\,\vec{x}\,\{Q\,\vec{x}\}) \to \\
& \mathbf{H}(\forall x_1\ldots\forall x_n\,\{P'\,\vec{x}\}\,c\,\vec{x}\,\{Q'\,\vec{x}\})
\end{aligned}
$$

A proof of this corollary can be obtained by directly translating half of the proof of Theorem 2.7 into LF. The example given there becomes the LF term

$$
\begin{aligned}
\lambda P,Q,R,c,d.\,\forall\text{IntroBin}\,&(\lambda x\,.\,\{P\,x\}\,c\,x\,\{Q\,x\}) \\
&(\lambda x\,.\,\{Q\,x\}\,d\,x\,\{R\,x\}) \\
&(\lambda x\,.\,\{P\,x\}\,c\,x\;;\,d\,x\,\{R\,x\}) \\
&(\lambda x\,.\,\text{SeqRule}_0\,(P\,x)\,(Q\,x)\,(R\,x)\,(c\,x)\,(d\,x))
\end{aligned}
$$

which has type

$$
\begin{aligned}
&\textstyle\prod_{P,Q,R:\mathbf{var}\to\mathbf{assert}} \prod_{c,d:\mathbf{var}\to\mathbf{comm}} \\
&\quad \mathbf{H}(\forall x\,\{P\,x\}\,c\,x\,\{Q\,x\}) \to \mathbf{H}(\forall x\,\{Q\,x\}\,d\,x\,\{R\,x\}) \to \mathbf{H}(\forall x\,\{P\,x\}\,c\,x\;;\,d\,x\,\{R\,x\})
\end{aligned}
$$

and is thereby worthy of the name SeqRule_1.

Proposition 3.4 *For any permutation σ on $1,\ldots,n$, with $n\geq 2$, the rule*

$$
\begin{aligned}
\text{Exch}_\sigma \;:\; & \textstyle\prod_{f:\overbrace{\mathbf{var}\to\ldots\to\mathbf{var}}^{n}\to\mathbf{spec}} \\
& \mathbf{H}(\forall x_1\ldots\forall x_n\,f\,\vec{x}) \to \mathbf{H}(\forall x_{\sigma(1)}\ldots\forall x_{\sigma(n)}\,f\,\vec{x})
\end{aligned}
$$

is derivable.

Proof. First, we derive $\text{Exch}_{n,m}$ for each $1\leq m<n\in\omega$, of type

$$
\begin{aligned}
\text{Exch}_{n,m} \;:\; & \textstyle\prod_{f:\overbrace{\mathbf{var}\to\ldots\to\mathbf{var}}^{n}\to\mathbf{spec}} \\
& \mathbf{H}(\forall x_1\ldots\forall x_n\,f\,\vec{x}) \to \\
& \mathbf{H}(\forall x_1\ldots\forall x_{m-1}\forall x_{m+1}\forall x_m\forall x_{m+2}\ldots\forall x_n\,f\,\vec{x}),
\end{aligned}
$$

by induction on n:

Base Case $\text{Exch}_{2,1}\stackrel{\text{dfn}}{=}\text{Exch}$

Induction Step We assume the terms $\text{Exch}_{n,m}$ for all $m:1\leq m<n$. Then

$$
\begin{aligned}
\text{Exch}_{n+1,m} &\stackrel{\text{dfn}}{=} \lambda f\,.\,\text{Exch}_{n,m}\,(\lambda\vec{x}\,.\,\forall y\,f\,\vec{x}\,y) \\
&\quad (\text{for all } m:1\leq m<n); \\
\text{Exch}_{n+1,n} &\stackrel{\text{dfn}}{=} \lambda f\,.\,\forall\text{IntroUn}\,(\lambda y\,.\,\forall x_1\ldots\forall x_n\,f\,y\,\vec{x}) \\
&\qquad\qquad (\lambda y\,.\,\forall x_1\ldots\forall x_{n-2}\forall x_n\forall x_{n-1}\,f\,y\,\vec{x}) \\
&\qquad\qquad (\lambda y\,.\,\text{Exch}_{n,n-1}\,(f\,y))
\end{aligned}
$$

Because we can decompose any permutation σ into a sequence of such transpositions (Bubblesort comes to mind), composing some of the terms derived above will yield Exch_σ. □

One might complain that the ubiquitous subscripts appearing on the names of primitive or derived inference rules present a greater bureaucratic headache than the non-interference judgments of [7]. In a real implementation, most of these subscripts could be inferred from the types of the arguments; in an implementation of Mason's system, the non-interference proof obligations at each use of the assignment axiom can be mechanically derived, but depend on a plethora of non-interference assumptions in the context. It is a matter of personal taste.

4 Extensions

4.1 Local Variables

The rule for local variable declarations is relatively easy to add. In Hoare Logic it looks like

$$\frac{\{P \wedge y = e\}\, c[y/x]\, \{Q\}}{\{P\}\, \mathtt{new}\ x := e\ ;\ c\, \{Q\}} \quad \text{where } y \text{ is not free in } P,\ e,\ c, \text{ or } Q$$

(see [1]). In LF it is rendered as

$$\begin{aligned}\mathrm{NewRule}_{1,1} : \ & \textstyle\prod_{P,Q:\mathbf{assert}} \prod_{c:\mathbf{var}\to\mathbf{comm}} \prod_{e:\mathbf{exp}} \\ & \mathbf{H}(\forall y\, \{P \wedge y\uparrow = e\}\, c\, y\, \{Q\}) \to \mathbf{H}(\{P\}\, \mathtt{new}\, e\, c\, \{Q\})\end{aligned}$$

where the new command constructor

$$\mathtt{new} : \mathbf{exp} \to (\mathbf{var} \to \mathbf{comm}) \to \mathbf{comm}$$

has been added to the signature. We have enforced the side condition of the informal rule by making the type of P and Q be **assert**, rather than **var** $\to$ **assert** as in the assignment axiom. This way, LF β-reduction prevents anything in P or Q from referring to the bound y.

Similarly to the proof of Proposition 3.4, one can derive

$$\begin{aligned}\mathrm{NewRule}_{n,m} : \ & \textstyle\prod_{P,Q:\overbrace{\mathbf{var}\to\ldots\to\mathbf{var}}^{n-1}\to\mathbf{assert}} \prod_{c:\overbrace{\mathbf{var}\to\ldots\to\mathbf{var}}^{n}\to\mathbf{comm}} \prod_{e:\overbrace{\mathbf{var}\to\ldots\to\mathbf{var}}^{n-1}\to\mathbf{exp}} \\ & \mathbf{H}(\forall x_1 \ldots \forall x_n\, \{(P\,(\vec{x} \setminus x_m)) \wedge x_m\uparrow = (e\,(\vec{x} \setminus x_m))\}\, c\, \vec{x}\, \{Q\,(\vec{x} \setminus x_m)\}) \to \\ & \mathbf{H}(\forall x_1 \ldots \forall x_{m-1} \forall x_{m+1} \ldots \forall x_n \\ & \quad \{P\,(\vec{x} \setminus x_m)\}\, \mathtt{new}\, (e\,(\vec{x} \setminus x_m))\, (\lambda x_m\, .\, c\, \vec{x})\, \{Q\,(\vec{x} \setminus x_m)\})\end{aligned}$$

for all $1 \le m \le n \in \omega$, where $(\vec{x} \setminus x_m)$ abbreviates $x_1 \ldots x_{m-1}\, x_{m+1} \ldots x_n$. If we add the informal equivalents of $\mathrm{NewRule}_{n,m}$ to QHL_0 and $\mathrm{NewRule}_{1,1}$ to QHL_1, we again have analogues of Proposition 2.5, Theorems 2.6 and 2.7, and Corollary 2.8. The adequacy theorems go through just as easily.

4.2 Data Types

Adding additional base types involves changing T into a multi-sorted theory and properly encoding it in Σ_T. Perhaps the cleanest technique is to add constants **DataType** : **TYPE** and δ : **DataType** for each data type δ, and modify the types of the two constants

$$\mathbf{var}, \mathbf{exp} : \mathbf{DataType} \to \mathbf{TYPE}.$$

The term formation constructors, axioms, and inference rules can take an additional parameter $D : \mathbf{DataType}$ describing the sorts of expressions to which they are being applied, as in

$$\begin{aligned} \forall \;&:\; \textstyle\prod_{D:\mathbf{DataType}}(\mathbf{var}(D) \to \mathbf{spec}) \to \mathbf{spec} \\ \mathrm{AsgnAx}_{1,1} \;&:\; \textstyle\prod_{D:\mathbf{DataType}} \prod_{P:\mathbf{exp}(D)\to\mathbf{assert}} \prod_{t:\mathbf{exp}(D)\to\mathbf{exp}(D)} \\ &\qquad \mathbf{H}(\forall_D\, x\, \{P(t\, x\uparrow_D)\}\, x :=_D t\, x\uparrow_D\, \{P\, x\uparrow_D\}). \end{aligned}$$

We will not use data types in the remainder of the paper.

4.3 Induction

The LF is an *open-ended* framework, meaning that one cannot automatically deduce that the only inhabitants of a certain type are those built from constructors in the signature. An example of this incompleteness as it pertains to our signature Σ^T_{QHL} is provided by the following proposition.

Proposition 4.1 *Under signature Σ^T_{QHL} and in the empty context, the LF type*

$$\mathrm{Vacuous} \stackrel{\mathrm{dfn}}{=} \textstyle\prod_{c:\mathbf{var}\to\mathbf{comm}} \mathbf{H}(\forall y\, \{y\uparrow \neq y\uparrow\}\, c\, y\, \{y\uparrow = y\uparrow\})$$

is not inhabited.

Proof (sketch). Suppose there were a term p such that $\vdash_{\mathrm{LF}} p : \mathrm{Vacuous}$. Without loss of generality assume p is in canonical form. Then

$$\begin{array}{lrcl} \vdash_{\mathrm{LF}} & p\,(\lambda y\,.\, y := y\uparrow) & : & \mathbf{H}(\forall y\, \{y\uparrow \neq y\uparrow\}\, y := y\uparrow\, \{y\uparrow = y\uparrow\}) \\ \vdash_{\mathrm{LF}} & p\,(\lambda y\,.\, y := y\uparrow\ ;\ y := y\uparrow) & : & \mathbf{H}(\forall y\, \{y\uparrow \neq y\uparrow\}\, y := y\uparrow\ ;\ y := y\uparrow\, \{y\uparrow = y\uparrow\}). \end{array}$$

Line 1 implies that p contains no occurrences of the rule $\mathrm{SeqRule}_0$, but line 2 implies that p contains exactly one occurrence of $\mathrm{SeqRule}_0$. □

Clearly the type Vacuous corresponds to an admissible (though not derivable) schema of QHL_1. The proof of admissibility depends on an induction over the form of the command c. We therefore add to Σ^T_{QHL} the rule constant

$$\begin{aligned} \mathrm{SpecInd}_1 :\; & \textstyle\prod_{f:(\mathbf{var}\to\mathbf{comm})\to\mathbf{spec}} \\ & \Big(\textstyle\prod_{t:\mathbf{exp}\to\mathbf{exp}} \mathbf{H}(f\,(\lambda x\,.\, x := t\, x\uparrow))\Big) \to \\ & \Big(\textstyle\prod_{c,d:\mathbf{var}\to\mathbf{comm}} \mathbf{H}(f\, c) \to \mathbf{H}(f\, d) \to \mathbf{H}(f\,(\lambda x\,.\, c\, x\ ;\ d\, x))\Big) \to \\ & \Big(\textstyle\prod_{b:\mathbf{exp}\to\mathbf{assert}} \prod_{s:(\prod_{y:\mathbf{exp}} \mathbf{QF}(b\, y))} \prod_{c:\mathbf{var}\to\mathbf{comm}} \\ & \quad \mathbf{H}(f\, c) \to \mathbf{H}(f\,(\lambda x\,.\, \mathtt{while}\,(b\, x\uparrow)\,(s\, x\uparrow)\,(c\, x)))\Big) \to \\ & \Big(\textstyle\prod_{b:\mathbf{exp}\to\mathbf{assert}} \prod_{s:(\prod_{y:\mathbf{exp}} \mathbf{QF}(b\, y))} \prod_{c,d:\mathbf{var}\to\mathbf{comm}} \\ & \quad \mathbf{H}(f\, c) \to \mathbf{H}(f\, d) \to \mathbf{H}(f\,(\lambda x\,.\, \mathtt{if}\,(b\, x\uparrow)\,(s\, x\uparrow)\,(c\, x)\,(d\, x)))\Big) \to \\ & \textstyle\prod_{c:\mathbf{var}\to\mathbf{comm}} \mathbf{H}(f\, c). \end{aligned}$$

The soundness of this rule follows from the characterization of canonical forms in LF [5, Lemma 2.10]. Using it we can easily construct a term of type Vacuous, and we can derive the Invariance Axiom

$$\textstyle\prod_{P:\mathbf{assert}} \prod_{c:\mathbf{var}\to\mathbf{comm}} \mathbf{H}(\forall x\, \{P\}\, c\, x\, \{P\}).$$

Note that the logic is still incomplete (i.e., it is not possible to derive all the admissible rules of QHL_1), for we cannot apply this rule to, say, higher-order judgments.

5 Examples

The reward for our endeavors arises from the full power of the LF as applied to the signature Σ^T_{QHL}; by identifying the LF λ and $\rightarrow$ with those of Idealized Algol [13] we can form terms (and corresponding proofs) of higher types. In this section we assume that T, the first-order theory relative to which the assertion calculus is defined, is Peano Arithmetic.

5.1 Factorial

A procedure for computing the factorial of a number i and assigning it to the variable f is

$$\begin{aligned}\mathrm{fac} \stackrel{\mathrm{dfn}}{=}\ & \lambda i.\,\lambda f.\,\mathtt{new}\, i\,(\lambda n.\, f := 1\ ; \\ & \qquad \mathtt{while}\,(\neg n\uparrow = 0)\,(\mathrm{QF}\neg\,(n\uparrow = 0)\,(\mathrm{QF}{=}\, n\uparrow\, 0)) \\ & \qquad\quad (f := f\uparrow \times n\uparrow\ ;\ n := n\uparrow - 1)) \\ \mathrm{fac}\ :\ & \mathbf{exp} \rightarrow \mathbf{var} \rightarrow \mathbf{comm}\end{aligned}$$

The following term, with reconstructible or assumed subterms elided to $\cdot$, proves the partial correctness of fac.

$$\begin{aligned}\mathrm{Fac} \stackrel{\mathrm{dfn}}{=}\ & \lambda R.\,\lambda i.\,\mathrm{NewRule}_{2,2}\,(\lambda f.\,R)\,(\lambda f.\,R \wedge f = i!)\cdot\cdot \\ & \quad (\mathrm{ConRule}_2 \cdot\cdot\cdot\cdot\cdot\cdot\cdot \\ & \qquad (\mathrm{SeqRule}_2 \cdot\cdot\cdot\cdot\cdot \\ & \qquad\quad (\mathrm{AsgnAx}_{2,1}\cdot\cdot) \\ & \qquad\quad (\mathrm{WhileRule}_2\,(\lambda f.\,\lambda n.\,R \wedge f \times n! = i!)\cdot\cdot\cdot \\ & \qquad\qquad (\mathrm{ConRule}_2 \cdot\cdot\cdot\cdot\cdot\cdot\cdot \\ & \qquad\qquad\quad (\mathrm{SeqRule}_2\,(\lambda f.\,\lambda n.\,R \wedge (f \times n) \times (n-1)! = i!)\cdot\cdot\cdot\cdot \\ & \qquad\qquad\qquad (\mathrm{AsgnAx}_{2,1}\cdot\cdot) \\ & \qquad\qquad\qquad (\mathrm{AsgnAx}_{2,2}\cdot\cdot)))))) \\ \mathrm{Fac}\ :\ & \textstyle\prod_{R:\mathbf{assert}} \prod_{i:\mathbf{exp}} \mathbf{H}(\forall f\,\{R\}\,\mathrm{fac}\,i\,f\,\{R \wedge f\uparrow = i!\})\end{aligned}$$

The assertion R, which appears in the pre- and post-condition, cannot access the bound variable f or the local variable n; thus we have shown something slightly stronger than is possible (formally) in Hoare Logic, namely that the active area of the factorial procedure is limited to the variable f. Furthermore, if we intend to reuse this proof (see the next section), R will enable us to enlarge the active area referred to by the pre- and post-condition.

Observe that the passive argument i is bound by $\prod$, whereas the active argument f must be bound by $\forall$ in order to construct the proof. This pattern occurs in general when reasoning about proper procedures (those whose return type is **comm**), in light of Proposition 2.5.

5.2 Choose

A procedure for computing m choose k and leaving the result in c is given by

$$\begin{aligned}\mathrm{choose} \stackrel{\mathrm{dfn}}{=}\ & \lambda m.\,\lambda k.\,\lambda c.\,\mathtt{new}\,0\,(\lambda a.\,\mathrm{fac}\,k\,c\ ; \\ & \qquad\qquad \mathrm{fac}\,(m-k)\,a\ ;\ c := c\uparrow \times a\uparrow\ ; \\ & \qquad\qquad \mathrm{fac}\,m\,a\ ;\ c := a\uparrow / c\uparrow). \\ \mathrm{choose}\ :\ & \mathbf{exp} \rightarrow \mathbf{exp} \rightarrow \mathbf{var} \rightarrow \mathbf{comm}\end{aligned}$$

We can specify the partial correctness of this procedure in terms of the proof Fac from the preceding section. Although the type of the proof term Choose below has only one $\forall$-bound

identifier, most of the proof is carried out inside of the bindings $\forall c \forall a$. The Fac proof shows only one binding, $\forall f$. In order to use Fac for the three calls to fac within the proof Choose, we will need to invoke structural rules $\forall$IntroAx and Exch to properly instantiate the active variable f, viz.,

$$\begin{aligned} \mathrm{Fac}_{2,1}\, R\, i &\stackrel{\mathrm{dfn}}{=} \mathrm{Exch} \cdot (\forall\mathrm{IntroAx} \cdot (\lambda c\,.\,\mathrm{Fac}\, R\, i)) &&: \mathbf{H}(\forall c \forall a\, \{R\}\, \mathrm{fac}\, i\, c\, \{R \wedge c\uparrow = i!\}) \\ \mathrm{Fac}_{2,2}\, R\, i &\stackrel{\mathrm{dfn}}{=} \forall\mathrm{IntroAx} \cdot (\lambda c\,.\,\mathrm{Fac}\, R\, i) &&: \mathbf{H}(\forall c \forall a\, \{R\}\, \mathrm{fac}\, i\, a\, \{R \wedge a\uparrow = i!\}). \end{aligned}$$

The skeleton of the proof term looks like

$$\begin{aligned} \mathrm{Choose} \stackrel{\mathrm{dfn}}{=}\ & \mathrm{NewRule}_{2,2} \cdot \cdot \cdot \cdot \cdot \\ & (\mathrm{SeqRule}_2 \cdot \cdot \cdot \cdot \cdot \\ & \quad (\mathrm{SeqRule}_2 \cdot \cdot \cdot \cdot \cdot \\ & \quad\quad (\mathrm{SeqRule}_2 \cdot \cdot \cdot \cdot \cdot \\ & \quad\quad\quad (\mathrm{SeqRule}_2 \cdot \cdot \cdot \cdot \cdot \\ & \quad\quad\quad\quad (\mathrm{ConRule}_2 \cdot \cdot \cdot \cdot \cdot \cdot \cdot (\mathrm{Fac}_{2,1} \cdot k)) \\ & \quad\quad\quad\quad (\mathrm{ConRule}_2 \cdot \cdot \cdot \cdot \cdot \cdot \cdot (\mathrm{Fac}_{2,2} \cdot (m - k)))) \\ & \quad\quad\quad (\mathrm{AsgnAx}_{2,1} \cdot \cdot)) \\ & \quad\quad (\mathrm{ConRule}_2 \cdot \cdot \cdot \cdot \cdot \cdot \cdot (\mathrm{Fac}_{2,2} \cdot m))) \\ & (\mathrm{AsgnAx}_{2,1} \cdot \cdot)) \end{aligned}$$

$$\mathrm{Choose} \ :\ \textstyle\prod_{m,k:\mathbf{exp}} \mathbf{H}(\forall c\, \{k \leq m\}\, \mathrm{choose}\, m\, k\, c \left\{ c\uparrow = \binom{m}{k} \right\}).$$

5.3 Interference Control

In the case of procedures with multiple active arguments, the built-in limitations of the structural rules provide a weak form of interference control. Suppose we have a proof

$$R : \textstyle\prod_{\vec{x}:\mathbf{exp}} \mathbf{H}(\forall \vec{y}\, \{P_r\, \vec{x}\, \vec{y}\uparrow\}\, r\, \vec{x}\, \vec{y}\, \{Q_r\, \vec{x}\, \vec{y}\uparrow\})$$

about procedure r whose active arguments are represented by $\vec{y}$ in R, and we wish to use R to construct a proof term

$$C : \textstyle\prod_{\vec{w}:\mathbf{exp}} \mathbf{H}(\forall \vec{z}\, \{P\, \vec{w}\, \vec{z}\uparrow\} \ldots r\, \vec{t}\, \vec{v} \ldots \{Q\, \vec{w}\, \vec{z}\uparrow\}),$$

where $\vec{v} \subseteq \vec{z}$ and $\vec{t}$ are all of type **exp**. We must apply structural rules to R to obtain a term

$$R' : \textstyle\prod_{\vec{x}:\mathbf{exp}} \mathbf{H}(\forall \vec{z}\, \{P_r\, \vec{x}\, \vec{v}\uparrow\}\, r\, \vec{x}\, \vec{v}\, \{Q_r\, \vec{x}\, \vec{v}\uparrow\}),$$

because the rules for compound commands which we will use to construct C expect identical prefixes in their antecedents. (If r occurs within k **new** commands, then k additional active identifiers will be needed in the prefix.) We can rename the identifiers in the prefix in the type of R by virtue of LF's definitional equality; the structural rules will permit arbitrary expansion and rearrangement of the prefix, but they provide no means for contraction to occur. Globals in the body of r can be treated as additional active parameters. Therefore $\vec{v}$ must all be distinct (i.e., non-interfering) identifiers, and $\vec{v}$ must not interfere with r, in order for us to use R in the construction of C. Finally, apply R' to the arguments $\vec{t}$ and abstract over $\vec{w}$ to obtain

$$\lambda \vec{w}\,.\, R'\, \vec{t} : \textstyle\prod_{\vec{w}:\mathbf{exp}} \mathbf{H}(\forall \vec{z} \left\{ P_r\, \vec{t}\, \vec{v}\uparrow \right\} r\, \vec{t}\, \vec{v} \left\{ Q_r\, \vec{t}\, \vec{v}\uparrow \right\})$$

which can be used as an antecedent in some compositional rule.

Reasoning about interfering identifiers is not completely precluded, however. For example, an easy modification to the proof term Fac of section 5.1 yields a term

$$\mathrm{Fac}' : \prod_{R:\mathbf{assert}} \prod_{i:\mathbf{exp}} \prod_{j:\mathbf{var}\to\mathbf{exp}} \mathbf{H}(\forall f\,\{R \wedge (j\ f) = i\}\,\mathrm{fac}\,(j\ f)\ f\,\{R \wedge f\!\uparrow = i!\})$$

expressing the correctness of fac even if its second argument interferes with its first.

6 Conclusions and Future Work

We have made a simple syntactic adjustment to Hoare Logic and demonstrated how its representation in a meta-logic gives us a more powerful programming logic for free. Our encoding improves upon those in the literature in both elegance and power, avoiding judgments concerning non-interference and the restriction to a fixed finite collection of program variables. We have, with relative ease, added extensions to the signature which handle local variables, data types, and inductively-derived admissible rules, further evidence that our approach is the right one. A machine implementation of the logic has been directly obtained from the representation and used to check the examples of this paper. Thus we have reaped the philosophical, theoretical, and practical benefits of formalization mentioned in the introduction.

Mason suggests in [7, pg. 12] that the LF is to blame for the complexities of formalizing Hoare Logic, as it does not have a direct means to represent call-by-value substitution. The work presented here is intended to indicate that the fault lies in Hoare Logic itself: plain Hoare Logic does not respect substitution, and even the ornate form of it given in our calculus QHL_1 does not admit contraction of active identifiers, a non-standard structural rule which conflicts with those of the LF. U. S. Reddy[1] has proposed the possibility of a dual solution to our affine quantifier $\forall$: formalizing Hoare Logic in an "affine LF" which disallows contraction at the meta-level, requiring the introduction of a classical quantifier $\mathbb{C}$ to handle passive identifier occurrences. Perhaps the cleanest encoding could be obtained in a version of LF which employed both kinds of dependent product as primitive, evoking the type regimen of Syntactic Control of Interference in [9].

Part of the "essence of Algol" [12] is the simplicity of its definition, which is essentially the inclusion of commands as a base type in a typed λ-calculus. It seems reasonable, therefore, to approach the design of a programming logic for Algol in the same way, namely as the inclusion of Hoare formulas as a basic judgment form in a logical framework consisting of a typed λ-calculus. We believe that our work is the first step towards such a logic. Two avenues of research in this vein present themselves:

1. We are investigating the consequences of internalizing phrase types, type constructors, and the notion of computation, and extending the programming language to include conditionals and recursion at arbitrary phrase types. It appears that the resulting system will be a sound logic for Algol which is cleaner than, although non-trivially related to, Specification Logic. See the appendix for a brief discussion. This approach actually reveals yet another benefit of formalization: we can concisely specify a *new* logic in the meta-logic, allowing the theory of the meta-logic to generate the object logic we want.

2. Strengthening the type system of the LF itself will lead to a correspondingly stronger programming logic. For example, there is a published formulation of LF which include pairing [4] which is believed to preserve its good properties. The closer the type theory of the meta-logic

[1] Private communication.

matches the type theory of Algol, the more attractive our formalization will become as a basis for a programming logic.

An additional challenge is presented in light of an unpublished paper of M. Miculan [8]. Standard encodings of first-order logic in LF are adequate for the consequence relation of *truth* rather than *validity* [2], yet the rules given for Hoare Logic are sound only with respect to validity. In particular, there exist proofs from assumptions in Hoare Logic which have no counterpart in the LF encoding. This fact does not contradict our adequacy theorem, which is stated with respect to the empty context only; however, we would like to incorporate the side conditions introduced in [8] to broaden the semantic adequacy of the encoding.

7 Acknowledgments

I wish to thank Uday S. Reddy for suggesting the problem, providing guidance and insight during its solution, and carefully reviewing early drafts of this paper.

References

[1] K. R. Apt, *Ten years of Hoare's Logic: A survey—part I*, ACM Transactions on Programming Languages and Systems, 3 (1981), pp. 431–483.

[2] A. Avron, *Simple consequence relations*, Information and Conputation, 92 (1991), pp. 105–139.

[3] A. Avron, F. Honsell, I. A. Mason, and R. Pollack, *Using typed λ-calculus to implement formal systems on a machine*, Journal of Automated Reasoning, 9 (1992), pp. 309–354.

[4] Y. Fu, *Categorical properties of logical frameworks*, Tech. Rep. UMCS-93-6-3, Department of Computer Science, Manchester University, Manchester, U.K., 1993.

[5] R. Harper, F. Honsell, and G. Plotkin, *A framework for defining logics*, Journal of the Association for Computing Machinery, 40 (1993), pp. 143–184.

[6] F. Honsell and M. Miculan, *A natural deduction approach to program logics.* Submitted to international workshop TYPES '94, 1994.

[7] I. A. Mason, *Hoare's Logic in the LF*, Tech. Rep. ECS-LFCS-87-32, Laboratory for the Foundations of Computer Science, Edinburgh, Scotland, 1987.

[8] M. Miculan, *The rôle of assumptions in Hoare's Logic.* Anonymous ftp to ftp.di.unipi.it, file pub/Papers/miculan/assumptions_HL.dvi.gz, Dec. 1992.

[9] P. W. O'Hearn and R. D. Tennent, *Syntactic control of interference revisited*, May 1994.

[10] J. C. Reynolds, *Syntactic control of interference*, in 5th Annual ACM Symposium on Principles of Programming Languages, Tuscon, Arizona, 1978, Association for Computing Machinery, pp. 39–46.

[11] ——, *The Craft of Programming*, Series in Computer Science, Prentice-Hall International, London, 1981.

[12] ——, *The essence of ALGOL*, in Algorithmic Languages, J. W. de Bakker and J. C. van Vliet, eds., North-Holland, 1981, pp. 345–372.

[13] ——, *Idealized ALGOL and its specification logic*, in Tools and Notions for Program Construction: An Advanced Course, D. Néel, ed., Cambridge University Press, 1982, pp. 121–161.

[14] ——, *Syntactic control of interference, part 2*, in International Colloquium on Automata, Languages, and Programming, G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, eds., vol. 372 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1989, pp. 704–722.

A Towards a Signature for Algol

The rules of the signature Σ^T_{HL}, modulo some syntactic dressing, can become the nucleus of a logic for full Idealized Algol. The only significant change we make is to internalize phrase types and the notion of computation, so that recursion can be encompassed.

A.1 Phrase Types

To encode the syntax of Algol we must internalize its type constructors. The syntax of the new signature, Σ^T_{Alg}, is shown in Table 5.

For simplicity, we have removed boolean expressions (quantifier-free assertions) from commands altogether. Instead we insist that there be a designated constant expression 0 and define the conditional if0 to test its first argument for equality to 0. (The theory T can internalize the boolean connectives and predicate symbols as function symbols so that no expressiveness is lost.) Thus recursively-defined expressions can appear in conditionals, but assertions cannot be recursively defined.

The new form of the conditional allows us to create "bad" variables [13]. A bad variable is a term of type **var** such that assigning value v to it and immediately dereferencing it may not yield v. The assignment axiom will need no modifications, however, because it only permits us to reason about assignments to $\forall$-bound identifiers in a prefix, which are assumed to denote good variables and cannot (through substitution) be replaced by bad ones. In fact, using the assignment axiom we can only reason about terms of type **var** which compute (see below) to one of the prefix identifiers; other such terms, like conditional variables depending on active values, must first invoke a rule like IfFnAx to be handled. By virtue of this soundness property, **new** creates only good variables.

A.2 Judgment Forms

Having removed assertions from commands we have no further use for **QF**. However, to reason about conditional expressions and phrases of higher type, we need another judgment form internalizing the notion of computation, similar to the $\equiv_\theta$ of Specification Logic or the $I(A, a, b)$ types of Martin-Löf type theory:

$$\begin{array}{rcl} \mathbf{comp} & : & \mathbf{TYPE} \\ \Rrightarrow & : & \prod_{T:\mathbf{PhrType}} \overline{T} \to \overline{T} \to \mathbf{comp} \\ \mathbf{C} & : & \mathbf{comp} \to \mathbf{TYPE}. \end{array}$$

A.3 Axioms and Inference Rules

Most of the rules from Σ^T_{QHL} carry over almost verbatim. We also must include computation rules and substitution rules for allowing computationally-equivalent terms to appear interchangeably in judgments. The complete set of rules is shown in Table 6, and the logic PL_{Alg} is defined to be the set of terms generated in LF by the signature Σ^T_{Alg}.

$\mathbf{spec}, \mathbf{assert}, \mathbf{PhrType}$	:	$\mathbf{TYPE}$	
$\overline{\ }$	:	$\mathbf{PhrType} \to \mathbf{TYPE}$	
$\mathbf{exp}, \mathbf{comm}$	:	$\mathbf{PhrType}$	
$\to, \times$	:	$\mathbf{PhrType} \to \mathbf{PhrType} \to \mathbf{PhrType}$	
$\mathbf{acc}$	$\stackrel{\text{dfn}}{=}$	$\mathbf{exp} \to \mathbf{comm} : \mathbf{PhrType}$	
$\mathbf{var}$	$\stackrel{\text{dfn}}{=}$	$\mathbf{exp} \times \mathbf{acc} : \mathbf{PhrType}$	
$\langle , \rangle$	:	$\prod_{S,T:\mathbf{PhrType}} \overline{S} \to \overline{T} \to \overline{S \times T}$	
π^l	:	$\prod_{S,T:\mathbf{PhrType}} \overline{S \times T} \to \overline{S}$	
π^r	:	$\prod_{S,T:\mathbf{PhrType}} \overline{S \times T} \to \overline{T}$	
λ	:	$\prod_{S,T:\mathbf{PhrType}} (\overline{S} \to \overline{T}) \to \overline{S \to T}$	
ap	:	$\prod_{S,T:\mathbf{PhrType}} \overline{S \to T} \to (\overline{S} \to \overline{T})$	
Y	:	$\prod_{T:\mathbf{PhrType}} \overline{T \to T} \to \overline{T}$	
if0	:	$\prod_{T:\mathbf{PhrType}} \overline{\mathbf{exp}} \to \overline{T} \to \overline{T} \to \overline{T}$	
$\bot$	$\stackrel{\text{dfn}}{=}$	$\lambda T \,.\, \mathrm{Y}_T \, (\lambda_{T,T} \, (\lambda x \,.\, x)) : \prod_{T:\mathbf{PhrType}} \overline{T}$	
$\{\_\} \_ \{\_\}$	:	$\mathbf{assert} \to \overline{\mathbf{comm}} \to \mathbf{assert} \to \mathbf{spec}$	
$\forall$	:	$(\overline{\mathbf{var}} \to \mathbf{spec}) \to \mathbf{spec}$	
$0, c$	:	$\overline{\mathbf{exp}}$	$[c \in T]$
f	:	$\overbrace{\overline{\mathbf{exp}} \to \ldots \to \overline{\mathbf{exp}}}^{n} \to \overline{\mathbf{exp}}$	$[f \in T, ar(f) = n]$
$=$	:	$\overline{\mathbf{exp}} \to \overline{\mathbf{exp}} \to \mathbf{assert}$	
r	:	$\overbrace{\overline{\mathbf{exp}} \to \ldots \to \overline{\mathbf{exp}}}^{m} \to \mathbf{assert}$	$[r \in T, ar(r) = m]$
$\Rightarrow, \wedge, \vee$	:	$\mathbf{assert} \to \mathbf{assert} \to \mathbf{assert}$	
$\neg$	:	$\mathbf{assert} \to \mathbf{assert}$	
$\forall, \exists$	:	$(\overline{\mathbf{exp}} \to \mathbf{assert}) \to \mathbf{assert}$	
$\uparrow$	$\stackrel{\text{dfn}}{=}$	$\pi^l_{\mathbf{exp},\mathbf{exp}\to\mathbf{comm}} : \overline{\mathbf{var}} \to \overline{\mathbf{exp}}$	
$\mathtt{skip}$	:	$\overline{\mathbf{comm}}$	
$:=$	$\stackrel{\text{dfn}}{=}$	$\mathrm{ap}_{\mathbf{exp},\mathbf{comm}} : \overline{\mathbf{acc}} \to \overline{\mathbf{exp}} \to \overline{\mathbf{comm}}$	
$;$	:	$\overline{\mathbf{comm}} \to \overline{\mathbf{comm}} \to \overline{\mathbf{comm}}$	
$\mathtt{while0}$	$\stackrel{\text{dfn}}{=}$	$\lambda e \,.\, \lambda c \,.\, \mathrm{Y}_{\mathbf{comm}} \, (\lambda d \,.\, \mathrm{if0}_{\mathbf{comm}} \, e \, (c \ ; \ d) \, \mathtt{skip})$	
	:	$\overline{\mathbf{exp}} \to \overline{\mathbf{comm}} \to \overline{\mathbf{comm}}$	
$\mathtt{new}$	:	$\overline{\mathbf{exp}} \to (\overline{\mathbf{var}} \to \overline{\mathbf{comm}}) \to \overline{\mathbf{comm}}$	

Table 5: Syntax of Σ^T_{Alg}.

$$\mathrm{AsgnAx}_{1,1} : \prod_{P:\overline{\mathbf{exp}}\to\mathbf{assert}} \prod_{t:\overline{\mathbf{exp}}\to\overline{\mathbf{exp}}} \mathbf{H}(\forall x\,\{P(t\,x\uparrow)\}\,x := t\,x\uparrow\,\{P\,x\uparrow\})$$

$$\mathrm{SkipAx}_0 : \prod_{P:\mathbf{assert}} \mathbf{H}(\{P\}\,\mathtt{skip}\,\{P\})$$

$$\mathrm{AbortAx}_0 : \prod_{P,Q:\mathbf{assert}} \mathbf{H}(\{P\}\,\bot_{\mathbf{comm}}\,\{Q\})$$

$$\mathrm{SeqRule}_0 : \prod_{P,Q,R:\mathbf{assert}} \prod_{c,d:\overline{\mathbf{comm}}} \mathbf{H}(\{P\}\,c\,\{Q\}) \to \mathbf{H}(\{Q\}\,d\,\{R\}) \to \mathbf{H}(\{P\}\,c\;;\;d\,\{R\})$$

$$\mathrm{IfRule}_0 : \prod_{P,Q:\mathbf{assert}} \prod_{e:\overline{\mathbf{exp}}} \prod_{c,d:\overline{\mathbf{comm}}} \mathbf{H}(\{P\wedge e=0\}\,c\,\{Q\}) \to \mathbf{H}(\{P\wedge e\neq 0\}\,d\,\{Q\}) \to \mathbf{H}(\{P\}\,\mathrm{if0}_{\mathbf{comm}}\,e\,c\,d\,\{Q\})$$

$$\mathrm{ConRule}_0 : \prod_{P',P,Q,Q':\mathbf{assert}} \prod_{c:\overline{\mathbf{comm}}} \mathbf{T}(P'\Rightarrow P) \to \mathbf{T}(Q\Rightarrow Q') \to \mathbf{H}(\{P\}\,c\,\{Q\}) \to \mathbf{H}(\{P'\}\,c\,\{Q'\})$$

$$\mathrm{NewRule}_{1,1} : \prod_{P,Q:\mathbf{assert}} \prod_{c:\mathbf{var}\to\mathbf{comm}} \prod_{e:\mathbf{exp}} \mathbf{H}(\forall x\,\{P\wedge x\uparrow = e\}\,c\,x\,\{Q\}) \to \mathbf{H}(\{P\}\,\mathtt{new}\,e\,c\,\{Q\})$$

$$\mathrm{Exch} : \prod_{f:\overline{\mathbf{var}}\to\overline{\mathbf{var}}\to\mathbf{spec}} \mathbf{H}(\forall x\forall y\,f\,x\,y) \to \mathbf{H}(\forall y\forall x\,f\,x\,y)$$

$$\forall\mathrm{IntroAx} : \prod_{f:\overline{\mathbf{var}}\to\mathbf{spec}} \left(\prod_{x:\overline{\mathbf{var}}} \mathbf{H}(f\,x)\right) \to \mathbf{H}(\forall\,f)$$

$$\forall\mathrm{IntroUn} : \prod_{f,g:\overline{\mathbf{var}}\to\mathbf{spec}} \Big(\prod_{x:\overline{\mathbf{var}}} \mathbf{H}(f\,x) \to \mathbf{H}(g\,x)\Big) \to \Big(\mathbf{H}(\forall\,f) \to \mathbf{H}(\forall\,g)\Big)$$

$$\forall\mathrm{IntroBin} : \prod_{f,g,h:\overline{\mathbf{var}}\to\mathbf{spec}} \Big(\prod_{x:\overline{\mathbf{var}}} \mathbf{H}(f\,x) \to \mathbf{H}(g\,x) \to \mathbf{H}(h\,x)\Big) \to \Big(\mathbf{H}(\forall\,f) \to \mathbf{H}(\forall\,g) \to \mathbf{H}(\forall\,h)\Big)$$

$$\mathrm{ReflAx} : \prod_{T:\mathbf{PhrType}} \prod_{a:\overline{T}} \mathbf{C}(a \Rrightarrow_T a)$$

$$\mathrm{ExpAx} : \prod_{t,t':\overline{\mathbf{exp}}} \mathbf{T}(t=t') \to \mathbf{C}(t \Rrightarrow_{\mathbf{exp}} t')$$

$$\mathrm{ApAx} : \prod_{S,T:\mathbf{PhrType}} \prod_{f:\overline{S\to T}} \prod_{a:\overline{S}} \mathbf{C}(((\lambda_{S,T}\,f)\,\mathrm{ap}_{S,T}\,a) \Rrightarrow_T (f\,a))$$

$$\mathrm{BotFnAx} : \prod_{S,T:\mathbf{PhrType}} \mathbf{C}(\bot_{S\to T} \Rrightarrow_{S\to T} \lambda_{S,T}\,(\lambda x\,.\,\bot_T))$$

$$\mathrm{BotProdAx} : \prod_{S,T:\mathbf{PhrType}} \mathbf{C}(\bot_{S\times T} \Rrightarrow_{S\times T} \langle \bot_S, \bot_T\rangle_{S,T})$$

$$\mathrm{ProjLAx} : \prod_{S,T:\mathbf{PhrType}} \prod_{a:\overline{S}} \prod_{b:\overline{T}} \mathbf{C}((\pi^l_{S,T}\ \langle a,b\rangle_{S,T}) \Rrightarrow_S a)$$

$$\mathrm{ProjRAx} : \prod_{S,T:\mathbf{PhrType}} \prod_{a:\overline{S}} \prod_{b:\overline{T}} \mathbf{C}((\pi^r_{S,T}\ \langle a,b\rangle_{S,T}) \Rrightarrow_T b)$$

$$\mathrm{IfFnAx} : \prod_{S,T:\mathbf{PhrType}} \prod_{e:\overline{\mathbf{exp}}} \prod_{f,g:\overline{S\to T}} \mathbf{C}((\mathrm{if0}_{S\to T}\,e\,f\,g) \Rrightarrow_T (\lambda_{S,T}\,(\lambda x\,.\,\mathrm{if0}_T\,e\,(f\,\mathrm{ap}_{S,T}\,x)\,(g\,\mathrm{ap}_{S,T}\,x))))$$

$$\mathrm{IfProdAx} : \prod_{S,T:\mathbf{PhrType}} \prod_{e:\overline{\mathbf{exp}}} \prod_{p,q:\overline{S\times T}} \mathbf{C}((\mathrm{if0}_{S\times T}\,e\,p\,q) \Rrightarrow_{S\times T} \Big\langle \mathrm{if0}_S\,e\,(\pi^l_{S,T}\,p)\,(\pi^l_{S,T}\,q), \mathrm{if0}_T\,e\,(\pi^r_{S,T}\,p)\,(\pi^r_{S,T}\,q)\Big\rangle_{S,T})$$

$$\mathrm{RecAx} : \prod_{T:\mathbf{PhrType}} \prod_{f:\overline{T\to T}} \mathbf{C}((\mathrm{Y}_T\,f) \Rrightarrow_T (f\,\mathrm{ap}_{T,T}\,(\mathrm{Y}_T\,f)))$$

$$\mathrm{FixInd} : \prod_{T:\mathbf{PhrType}} \prod_{f:\overline{T}\to\mathbf{spec}} \prod_{p:\overline{T\to T}} \mathbf{H}(f\,\bot_T) \to \Big(\prod_{t:\overline{T}} \mathbf{H}(f\,t) \to \mathbf{H}(f\,(p\,\mathrm{ap}_{T,T}\,t))\Big) \to \mathbf{H}(f\,(\mathrm{Y}_T\,p))$$

$$\mathrm{Subst}_{\mathbf{C}} : \prod_{T:\mathbf{PhrType}} \prod_{t,t':\overline{T}} \prod_{f:\overline{T}\to\mathbf{comp}} \mathbf{C}(t \Rrightarrow_T t') \to \mathbf{C}(f\,t') \to \mathbf{C}(f\,t)$$

$$\mathrm{Subst}_{\mathbf{T}} : \prod_{T:\mathbf{PhrType}} \prod_{t,t':\overline{T}} \prod_{f:\overline{T}\to\mathbf{assert}} \mathbf{C}(t \Rrightarrow_T t') \to \mathbf{T}(f\,t') \to \mathbf{T}(f\,t)$$

$$\mathrm{Subst}_{\mathbf{H}} : \prod_{T:\mathbf{PhrType}} \prod_{t,t':\overline{T}} \prod_{f:\overline{T}\to\mathbf{spec}} \mathbf{C}(t \Rrightarrow_T t') \to \mathbf{H}(f\,t') \to \mathbf{H}(f\,t)$$

Table 6: Axioms and Rules of $\Sigma^T_{\mathbf{Alg}}$.

An Imperative Object Calculus
Basic Typing and Soundness

Martín Abadi and Luca Cardelli

Digital Equipment Corporation, Systems Research Center

Abstract

We develop an imperative calculus of objects that is both tiny and expressive. Our calculus provides a minimal setting in which to study the operational semantics and the typing rules of object-oriented languages. We prove type soundness using a simple subject-reduction approach.

1. Introduction

Procedural languages are generally well-understood; their constructs are by now standard, and their formal underpinnings are solid. The fundamental features of procedural languages have been distilled into formalisms that prove useful in identifying and explaining issues of implementation, static analysis, semantics, and verification.

An analogous understanding has not yet emerged in object-oriented programming. There is no widespread agreement on the choice of basic constructs and on their properties. Consequently, practical object-oriented languages support many features and programming techniques, often with little concern for orthogonality.

With the aim of clarifying the fundamental features of object-oriented languages, we introduce a tiny but expressive imperative calculus. The calculus comprises objects, method invocation, method update, object cloning, and local definitions. In a quest for minimality, we take objects to be just collections of methods. Fields are important too, but they can be seen as a derived concept; for example a field can be viewed as a method that does not use its self parameter.

When fields and methods are identified it is trivial to convert one into the other, conceptually turning passive data into active computation and vice versa. The hiding of fields from public view has been widely advocated as a means of concealing representation choices, and thereby allowing flexibility in implementation. Identifying fields with methods confers much of the same flexibility, by eliminating fields.

The unification of fields with methods has also the advantage of simplicity. Both objects and object operations assume a uniform structure. In contrast, the separation of fields from methods induces a corresponding separation of object operations, and leads to the implicit or explicit splitting of objects into two components. Unifying fields with methods gives more compact and therefore more elegant calculi.

This unification, however, has one debatable consequence. The natural operation on methods is method invocation, and the natural operations on fields are field selection and field update. By unifying fields with methods, we can collapse field selection and method invocation into a single operation. To complete the unification, though, we are forced to generalize field update to method update.

The reliance on method update is one of the most unusual aspects of our formal treatment: this operation is not normally found in programming languages. However, method update can be seen as a form of dynamic inheritance [25] which is a feature found in object-based languages [7] but not yet in class-based languages [6]. Like other forms of dynamic inheritance, method update supports the dynamic modification of object behavior allowing objects, in a sense, to change their class dynamically. Thus, method update gives us an edge in modeling object-based constructions, in addition to allowing us to model the more traditional class-based constructions where fields and methods are sharply separated.

A further justification for method update can be found in the desire to tame dynamic inheritance. Dynamic inheritance has potentially unpredictable effects, due to the updating of shared state. These concerns have led to the search for better-behaved, restricted, dynamic inheritance mechanisms [23]. Method update is one of these better-behaved mechanisms, especially in the absence of delegation, as in our calculus. Method update is statically typable, and can be used to emulate the mode-switching applications of dynamic inheritance [13]. With method update we avoid some dangerous aspects of dynamic inheritance [14, 23], while maintaining its dynamic specialization aspects originally advocated by the Treaty of Orlando [22].

In this paper, we study an untyped calculus (section 2), and then we present a type structure for it (section 3). The only type constructor is one for object types: an object type is a list of method names and method result types. A subtyping relation between object types supports object subsumption, which allows an object to be used where an object with fewer methods is expected. We prove the consistency of our rules using a subject-reduction approach (section 4). Our technique is an extension of Harper's [16], using closures and stacks instead of formal substitutions. This approach yields a manageable proof for a realistic implementation strategy.

Elsewhere we have considered functional calculi [2-4]. The main novelty here is the treatment of imperative features, with corresponding proof techniques. In further work [5] we treat second-order type structures (with Self types) for an imperative calculus.

A few other object formalisms have been defined and studied. Many of these rely on purely functional models, with an emphasis on types [1, 8, 10, 12, 17, 19-21, 26]. Others deal with imperative features in the context of concurrency; see for example [28]. The works most closely related to ours are that of Eifrig *et al.* on LOOP [15] and that of Bruce and van Gent on TOIL [9]. LOOP and TOIL are typed, imperative, object-oriented languages with procedures, objects, and classes. Both take procedures, objects, and classes as primitive, with fairly complex rules; they also distinguish methods from fields. LOOP is translated into a somewhat simpler calculus, which includes record, function, reference, recursive, and F-bounded types. Our calculus is centered entirely on objects: procedures and classes can be defined from them. The collections of programs that can be written and typed in these formalisms are different. In spite of this, we all share the goal of modeling imperative object-oriented languages by precise semantic structures and sound type systems.

2. An Untyped Imperative Calculus

We begin with the syntax of an untyped imperative calculus. The initial syntax is minimal, but in sections 2.2 and 2.3 we show how to express convenient constructs such as fields and procedures. We omit how to encode basic data types, control structures, and classes, which can be treated much as in [4]. In section 2.5 we give an operational semantics.

2.1 Syntax

Syntax of the imp-ς calculus

a,b ::=	term	
x	variable	
$[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$	object	(l_i distinct)
a.l	method invocation	
$a.l \Leftarrow \varsigma(x)b$	method update	
clone(a)	cloning	
let x = a in b	let	

An object is a collection of components $l_i=\varsigma(x_i)b_i$, for distinct labels l_i and associated methods $\varsigma(x_i)b_i$; the order of these components does not matter, even for our deterministic operational semantics. The letter ς (sigma) is used as a binder for the self parameter of a method; ς(x)b is a method with self parameter x, to be bound to the host object, and body b.

A method invocation o.l results in the evaluation of the body of the method named l, with o bound to the self parameter.

A method update $o.l \Leftarrow \varsigma(y)b$ replaces the method named l with ς(y)b in o, and returns the modified object.

A cloning operation clone(o) produces a new object with the same labels as o, with each component sharing the methods of the corresponding component of o.

The let construct evaluates a term, binds it to a variable, and then evaluates a second term with that variable in scope. Sequential evaluation can be defined from let, by:

$a;b \triangleq \text{let } x=a \text{ in } b,$

for $x \notin FV(b)$.

2.2 Fields

In our **imp-ς** calculus, every component of an object contains a method. However, we can encode fields with eagerly evaluated contents by using the let construct. We write $[l_i=b_i\ ^{i\in 1..n}, l_j=\varsigma(x_j)b_j\ ^{j\in 1..m}]$ for an object where $l_i=b_i$ are fields and $l_j=\varsigma(x_j)b_j$ are methods. We also write a.l:=b for field update, and a.l, as before, for field selection. We abbreviate:

Encoding of fields

$[l_i=b_i\ ^{i\in 1..n}, l_j=\varsigma(x_j)b_j\ ^{j\in 1..m}]$		for $y_i \notin FV(b_i\ ^{i\in 1..n}, b_j\ ^{j\in 1..m})$, y_i distinct, i∈0..n
	$\triangleq$ let $y_1=b_1$ in ... let $y_n=b_n$ in $[l_i=\varsigma(y_0)y_i\ ^{i\in 1..n}, l_j=\varsigma(x_j)b_j\ ^{j\in 1..m}]$	
a.l:=b	$\triangleq$ let $y_1=a$ in let $y_2=b$ in $y_1.l \Leftarrow \varsigma(y_0)y_2$	for $y_i \notin FV(b)$, y_i distinct, i∈0..2

The semantics of an object with fields may depend on the order of its components, because of side-effects in computing contents of fields. The encoding specifies an evaluation order.

By an update, a method can be changed into a field and vice versa. Thus, we use somewhat interchangeably the names selection and invocation.

2.3 Procedures

The **imp-ς** calculus is so minimal that it does not include procedures, but these can be expressed too. We begin by considering informally a call-by-value λ-calculus with side-effects, **imp-λ**, that includes abstraction, application, and assignment to λ-bound variables. For example, assuming arithmetic primitives, (λ(x) x:=x+1; x)(3) is an **imp-λ** term yielding 4. We translate **imp-λ** into **imp-ς**.

Translation of procedures

$\langle\!\langle x \rangle\!\rangle_\rho$	≜	ρ(x) if x∈dom(ρ), and x otherwise
$\langle\!\langle \lambda(x)b \rangle\!\rangle_\rho$	≜	[arg = ς(x)x.arg, val = ς(x)$\langle\!\langle b \rangle\!\rangle_{\rho\{x \leftarrow x.arg\}}$]
$\langle\!\langle b(a) \rangle\!\rangle_\rho$	≜	(clone($\langle\!\langle b \rangle\!\rangle_\rho$).arg:=$\langle\!\langle a \rangle\!\rangle_\rho$).val
$\langle\!\langle x:=a \rangle\!\rangle_\rho$	≜	x.arg:=$\langle\!\langle a \rangle\!\rangle_\rho$

In the translation, an environment ρ maps each variable x either to x.arg if x is λ-bound, or to x if x is a free variable. A λ-abstraction is translated to an object with an arg component, for storing the argument, and a val method, for executing the body. The arg component is initially set to a divergent method, and is filled with an argument upon procedure call. A call activates the val method that can then access the argument through self as x.arg. An assignment x:=a updates x.arg, where the argument is stored (assuming that x is λ-bound). A procedure needs to be cloned when it is called; the clone provides a fresh location in which to store the argument of the call, preventing interference with other calls of the same procedure. Such interference would derail recursive invocations. (This encoding has similarities with the mechanism of method activation in the Self language [25].)

2.4 A Small Example

We give a trivial example as a notation drill. We use fields, procedures, and basic data types in defining a memory cell with get, set, and dup (duplicate) components:

```
let m = [get = 0,  set = ς(self) λ(b) self.get:=b,  dup = ς(self) clone(self)]
in m.set(1); m.get                                                  yields 1
```

This cell can be used as a prototype for building cells, which can then be customized by method update. For example, we may create a cell that accepts only non-negative integers through the set method:

```
let m = [get = 0,  set = ς(self) λ(b) self.get:=b,  dup = ς(self) clone(self)]
in m.dup.set ⇐ ς(self) λ(b) if b<0 then self.get:=0 else self.get:=b
```

2.5 Operational Semantics

We now give an operational semantics that relates terms to results in a global store. Object terms reduce to results consisting of sequences of store locations, one location for each object component. In order to stay close to standard implementation techniques, we avoid using formal substitutions during reduction. We describe a semantics based on stacks and closures. A stack $\mathcal{S}$ associates variables with results; a closure $\langle \varsigma(x)b, \mathcal{S} \rangle$ is a pair of a method together with a stack that is used for the reduction of the method body. A store maps locations to method closures; we write stores in the form $\iota_i \mapsto \langle \varsigma(x_i)b_i, \mathcal{S}_i \rangle^{\,i \in 1..n}$; we write $\sigma.\iota \leftarrow m$ for the result of putting m in the ι location of σ.

The operational semantics is expressed in terms of a relation that relates a store σ, a stack $\mathcal{S}$, a term b, a result v, and another store σ'. This relation is written $\sigma \bullet \mathcal{S} \vdash b \rightsquigarrow v \bullet \sigma'$, and it means that

with the store σ and the stack $\mathcal{S}$, the term b reduces to a result v, yielding an updated store σ'. The stack does not change. The operational semantics is presented formally as follows.

Operational semantics

ι			store location	(e.g., an integer)
v	::=	$[l_i=\iota_i\ ^{i\in 1..n}]$	result	(l_i distinct)
σ	::=	$\iota_i\mapsto(\varsigma(x_i)b_i,\mathcal{S}_i)\ ^{i\in 1..n}$	store	(ι_i distinct)
$\mathcal{S}$	::=	$x_i\mapsto v_i\ ^{i\in 1..n}$	stack	(x_i distinct)

Well-formed store judgment: $\sigma \vdash \diamond$

(Store ø)

$$\frac{}{\emptyset \vdash \diamond}$$

(Store ι)

$$\frac{\sigma\cdot\mathcal{S}\vdash\diamond \quad \iota\notin dom(\sigma)}{\sigma,\ \iota\mapsto\langle\varsigma(x)b,\mathcal{S}\rangle\vdash\diamond}$$

Well-formed stack judgment: $\sigma\cdot\mathcal{S}\vdash\diamond$

(Stack ø)

$$\frac{\sigma\vdash\diamond}{\sigma\cdot\emptyset\vdash\diamond}$$

(Stack x)

$$\frac{\sigma\cdot\mathcal{S}\vdash\diamond \quad \iota_i\in dom(\sigma) \quad x\notin dom(\mathcal{S}) \quad l_i,\iota_i\ \text{distinct} \quad \forall i\in 1..n}{\sigma\cdot\mathcal{S},\ x\mapsto[l_i=\iota_i\ ^{i\in 1..n}]\vdash\diamond}$$

Term reduction judgment: $\sigma\cdot\mathcal{S}\vdash a\rightsquigarrow v\cdot\sigma'$

(Red x)

$$\frac{\sigma\cdot\mathcal{S}',x\mapsto v,\mathcal{S}''\vdash\diamond}{\sigma\cdot\mathcal{S}',x\mapsto v,\mathcal{S}''\vdash x\rightsquigarrow v\cdot\sigma}$$

(Red Object)

$$\frac{\sigma\cdot\mathcal{S}\vdash\diamond \quad \iota_i\notin dom(\sigma) \quad \iota_i\ \text{distinct} \quad \forall i\in 1..n}{\sigma\cdot\mathcal{S}\vdash[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]\rightsquigarrow[l_i=\iota_i\ ^{i\in 1..n}]\cdot(\sigma,\ \iota_i\mapsto\langle\varsigma(x_i)b_i,\mathcal{S}\rangle\ ^{i\in 1..n})}$$

(Red Select)

$$\frac{\sigma\cdot\mathcal{S}\vdash a\rightsquigarrow[l_i=\iota_i\ ^{i\in 1..n}]\cdot\sigma' \quad \sigma'(\iota_j)=\langle\varsigma(x_j)b_j,\mathcal{S}'\rangle \quad x_j\notin dom(\mathcal{S}') \quad j\in 1..n \quad \sigma'\cdot\mathcal{S}',\ x_j\mapsto[l_i=\iota_i\ ^{i\in 1..n}]\vdash b_j\rightsquigarrow v\cdot\sigma''}{\sigma\cdot\mathcal{S}\vdash a.l_j\rightsquigarrow v\cdot\sigma''}$$

(Red Update)

$$\frac{\sigma\cdot\mathcal{S}\vdash a\rightsquigarrow[l_i=\iota_i\ ^{i\in 1..n}]\cdot\sigma' \quad j\in 1..n \quad \iota_i\in dom(\sigma')}{\sigma\cdot\mathcal{S}\vdash a.l_j\Leftarrow\varsigma(x)b\rightsquigarrow[l_i=\iota_i\ ^{i\in 1..n}]\cdot\sigma'.\iota_j\leftarrow\langle\varsigma(x)b,\mathcal{S}\rangle}$$

(Red Clone)

$$\frac{\sigma\cdot\mathcal{S}\vdash a\rightsquigarrow[l_i=\iota_i\ ^{i\in 1..n}]\cdot\sigma' \quad \iota_i\in dom(\sigma') \quad \iota'_i\notin dom(\sigma') \quad \iota'_i\ \text{distinct} \quad \forall i\in 1..n}{\sigma\cdot\mathcal{S}\vdash clone(a)\rightsquigarrow[l_i=\iota'_i\ ^{i\in 1..n}]\cdot(\sigma',\ \iota'_i\mapsto\sigma'(\iota_i)\ ^{i\in 1..n})}$$

(Red Let)

$$\frac{\sigma\cdot\mathcal{S}\vdash a\rightsquigarrow v'\cdot\sigma' \quad \sigma'\cdot\mathcal{S},\ x\mapsto v'\vdash b\rightsquigarrow v''\cdot\sigma''}{\sigma\cdot\mathcal{S}\vdash \text{let } x=a \text{ in } b\rightsquigarrow v''\cdot\sigma''}$$

A variable reduces to the result it denotes in the current stack. An object reduces to a result consisting of a fresh collection of locations; the store is extended to associate method closures to those locations. A selection operation reduces its object to a result, and activates the appropriate method closure. An update operation reduces its object to a result, and updates the appropriate store loca-

tion with a new method closure. A clone operation reduces its object to a result; then it allocates a fresh collection of locations that are associated to the existing method closures from the object. A let construct reduces to the result of reducing its body in a stack extended with the bound variable and the result of its associated term.

We illustrate reduction with two examples. The first one is a simple terminating reduction for the term $[l = \varsigma(x)[]].l$. The following represents a (partial) derivation tree, with bracketed subtrees:

$$\begin{array}{ll} \left[\begin{array}{ll} \emptyset\bullet\emptyset \vdash [l=\varsigma(x)[]] \rightsquigarrow [l=0]\bullet(0\mapsto\langle\varsigma(x)[],\emptyset\rangle) & \text{by (Red Object)} \\ (0\mapsto\langle\varsigma(x)[],\emptyset\rangle)\bullet(x\mapsto[l=0]) \vdash [] \rightsquigarrow []\bullet(0\mapsto\langle\varsigma(x)[],\emptyset\rangle) & \text{by (Red Object)} \end{array}\right. & \\ \emptyset\bullet\emptyset \vdash [l=\varsigma(x)[]].l \rightsquigarrow []\bullet(0\mapsto\langle\varsigma(x)[],\emptyset\rangle) & \text{by (Red Select)} \end{array}$$

We illustrate method overriding, and the creation of loops through the store, by evaluating the term $[l = \varsigma(x)\ x.l \Leftarrow \varsigma(y)x].l$.

let $\sigma_0 \equiv 0\mapsto\langle\varsigma(x)x.l\Leftarrow\varsigma(y)x, \emptyset\rangle$ and $\sigma_1 \equiv 0\mapsto\langle\varsigma(y)x, (x\mapsto[l=0])\rangle$

$$\begin{array}{ll} \left[\begin{array}{ll} \emptyset\bullet\emptyset \vdash [l=\varsigma(x)x.l\Leftarrow\varsigma(y)x] \rightsquigarrow [l=0]\bullet\sigma_0 & \text{by (Red Object)} \\ \quad\left[\ \sigma_0\bullet(x\mapsto[l=0]) \vdash x \rightsquigarrow [l=0]\bullet\sigma_0 \right. & \text{by (Red x)} \\ \sigma_0\bullet(x\mapsto[l=0]) \vdash x.l\Leftarrow\varsigma(y)x \rightsquigarrow [l=0]\bullet\sigma_1 & \text{by (Red Update)} \end{array}\right. & \\ \emptyset\bullet\emptyset \vdash [l=\varsigma(x)x.l\Leftarrow\varsigma(y)x].l \rightsquigarrow [l=0]\bullet\sigma_1 & \text{by (Red Select)} \end{array}$$

The store σ_1 contains a loop, because it maps the index 0 to a closure that binds the variable x to a value that contains index 0. Hence, an attempt to read out the result of $[l=\varsigma(x)x.l\Leftarrow\varsigma(y)x].l$ by "inlining" the store and stack mappings would produce the infinite term $[l=\varsigma(y)[l=\varsigma(y)[l=\varsigma(y)...]]]$.

The potential for creating loops in the store is characteristic of imperative semantics. Loops in the store complicate reasoning about programs and, as we see in the next chapter, they also demand special attention in the treatment of type soundness.

3. Typing

We define a type system for the untyped calculus of section 2, and give a typed example.

3.1 Typing Rules

The typing rules for objects are the same ones we would have for a functional semantics. They are in fact a superset of those of [4], except that terms do not contain type annotations (to match our untyped operational semantics).

Typing rules

Well-formed environment and type judgments: $E \vdash \diamond$, $E \vdash A$

(Env ∅)
$$\frac{}{\emptyset \vdash \diamond}$$

(Env x)
$$\frac{E \vdash A \qquad x \notin dom(E)}{E,x:A \vdash \diamond}$$

(Type Object) (l_i distinct)
$$\frac{E \vdash B_i \qquad \forall i \in 1..n}{E \vdash [l_i:B_i\ ^{i\in 1..n}]}$$

Subtyping judgment: $E \vdash A <: B$

(Sub Refl) $\dfrac{E \vdash A}{E \vdash A <: A}$ (Sub Trans) $\dfrac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C}$ (Sub Object) (l_i distinct) $\dfrac{E \vdash B_i \quad \forall i \in 1..n+m}{E \vdash [l_i:B_i\ ^{i \in 1..n+m}] <: [l_i:B_i\ ^{i \in 1..n}]}$

Value typing judgment: $E \vdash a : A$

(Val Subsumption) $\dfrac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$ (Val x) $\dfrac{E', x:A, E'' \vdash \diamond}{E', x:A, E'' \vdash x:A}$ (Val Object) (where $A \equiv [l_i:B_i\ ^{i \in 1..n}]$) $\dfrac{E, x_j:A \vdash b_j : B_j \quad \forall j \in 1..n}{E \vdash [l_i = \varsigma(x_i)b_i\ ^{i \in 1..n}] : A}$

(Val Select) $\dfrac{E \vdash a : [l_i:B_i\ ^{i \in 1..n}] \quad j \in 1..n}{E \vdash a.l_j : B_j}$ (Val Update) (where $A \equiv [l_i:B_i\ ^{i \in 1..n}]$) $\dfrac{E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x)b : A}$

(Val Clone) (where $A \equiv [l_i:B_i\ ^{i \in 1..n}]$) $\dfrac{E \vdash a : A}{E \vdash \text{clone}(a) : A}$ (Val Let) $\dfrac{E \vdash a : A \quad E, x:A \vdash b : B}{E \vdash \text{let } x = a \text{ in } b : B}$

The first two groups of rules concern typing environments, types, and the subtyping relation. An object type is a collection of method names and associated method result types. A longer object type is a subtype of a shorter one, without variation in the common type components. The final group concerns typing of values. There is one rule for each construct in the calculus; in addition, a subsumption rule connects value typing with subtyping judgments.

3.2 A Typed Example

This section illustrates how to type a simple imperative example: movable points. In this example we rely on fields and procedures, as encoded in section 2. The encoding of procedures typechecks with $A \rightarrow B$ translated as [arg:A, val:B].

Trivial as it may seem, the example of movable points has been a notorious source of difficulties in functional settings (see [4]). These difficulties have resulted in the use of sophisticated type theories. In an imperative setting, however, some of these difficulties can be avoided altogether.

Consider one-dimensional and two-dimensional points, with coordinate fields (x and y) and methods that modify these fields (mv_x and mv_y). The coordinates are integers. We assume that integers and reals are available, perhaps through an encoding. For example, the origin points are:

$$p_1 \triangleq [x = 0,\ mv\_x = \varsigma(s)\ \lambda(dx)\ s.x := s.x + dx]$$

$$p_2 \triangleq [x = 0,\ y = 0,\ mv\_x = \varsigma(s)\ \lambda(dx)\ s.x := s.x + dx,\ mv\_y = \varsigma(s)\ \lambda(dy)\ s.y := s.y + dy]$$

In the type system of section 3.1, p_1 and p_2 can be given the types:

$$P_1 \triangleq [x:\text{Int},\ mv\_x:\text{Int} \rightarrow []]$$

$$P_2 \triangleq [x,y:\text{Int},\ mv\_x,mv\_y:\text{Int} \rightarrow []]$$

where P_2 is a subtype of P_1. This result type [] is obtained by subsumption. The imperative operational semantics produces the desired effect of moving a point, without requiring any particular result type for move methods. In contrast, in a functional framework an informative result type is necessary.

Imperatively, there is no loss of type information when moving a point. For example, suppose that f is defined with one-dimensional points in mind, with the type $P_1 \rightarrow []$, and $norm_2$ is defined for two-dimensional points, with the type $P_2 \rightarrow$Real:

$$f: P_1 \rightarrow [] \triangleq \lambda(p)\ p.mv\_x(1)$$
$$norm_2: P_2 \rightarrow \text{Real} \triangleq \lambda(p)\ sqrt(p.x\hat{}2 + p.y\hat{}2)$$

Since P_2 is a subtype of P_1, $f(p_2)$ is a legal call for $p_2:P_2$. Therefore, the following code typechecks and, as expected, returns 1:

$$f(p_2);\ norm_2(p_2)$$

Thus, we have applied a P_1 procedure to a P_2 point, and after this we are still able to use the point as a member of P_2. In contrast, in a functional setting we may try to write $norm_2(f(p_2))$, which is not well-typed if $f:P_1 \rightarrow []$.

Even in an imperative setting, however, it is common to define methods that produce new objects, as opposed to modifying existing ones. If mv_x is to return a new object, one must declare it with result type P_1 or P_2, to take advantage of any change to the x coordinate. One may try to redefine P_1 and P_2 as recursive types (for example, $P_1 \triangleq \mu(X)[x:\text{Int},\ mv\_x:\text{Int} \rightarrow X]$), but then P_2 is not a subtype of P_1. With this definition, all the typing difficulties common in functional settings resurface.

4. Soundness

We show the type soundness of our operational semantics, using an approach similar to subject reduction. We build on the techniques developed by Tofte, Wright and Felleisen, Leroy, and Harper [16, 18, 24, 27]. Our proof technique is an extension of Harper's, in that we deal with closures and stacks and thus avoid introducing locations into the term language.

The typing of results with respect to stores is delicate. We would not be able to determine the type of a result by examining its substructures recursively, including the ones accessed through the store, because stores may contain loops. Store types, introduced below, allow us to type results independently of particular stores. This is possible because type-sound computations do not store results of different types in the same location. Next we formalize store types and other notions necessary for the proof of soundness.

A store type Σ associates a method type to each store location. A method type has the form $[l_i:B_i\ ^{i \in 1..n}] \Rightarrow B_j$, where $[l_i:B_i\ ^{i \in 1..n}]$ is the type of self, and B_j is the result type, for $j \in 1..n$. The statement of soundness relies on a new judgment, result typing: $\Sigma \vDash v : A$. This means that the result v has type A with respect to the store type Σ. The locations contained in v are assigned types in Σ.

To connect stores and store types, we use a judgment $\Sigma \vDash \sigma$. Checking this judgment reduces to checking that the contents of every store location has the type determined by the store type for that location. Since locations contain closures and store types contain method types, we need to determine when a closure has a method type. For this, it is sufficient to check that a stack is compatible with an environment; the environment is then used to type the method. We write $\Sigma \vDash \mathcal{S} : E$ to mean that the stack $\mathcal{S}$ is compatible with the environment E in Σ. Now, since stacks contain results and environments contain types, we can define $\Sigma \vDash \mathcal{S} : E$ via the result typing judgment, which we have already discussed. The rule for store typing deals with each closure with respect to the whole store, accounting for cycles in the store.

Store typing rules

$M ::= [l_i:B_i\ ^{i\in 1..n}]\Rightarrow B_j$	method type	$(j\in 1..n)$
$\Sigma ::= \iota_i\mapsto M_i\ ^{i\in 1..n}$	store type	(ι_i distinct)
$\Sigma_1(\iota) \triangleq [l_i:B_i\ ^{i\in 1..n}]$	if	$\Sigma(\iota) = [l_i:B_i\ ^{i\in 1..n}]\Rightarrow B_j$
$\Sigma_2(\iota) \triangleq B_j$	if	$\Sigma(\iota) = [l_i:B_i\ ^{i\in 1..n}]\Rightarrow B_j$

Well-formed method type and store type judgments: $\vDash M \in \text{Meth},\ \Sigma \vDash \diamond$

(Method Type)

$$\frac{j\in 1..n}{\vDash [l_i:B_i\ ^{i\in 1..n}]\Rightarrow B_j \in \text{Meth}}$$

(Store Type)

$$\frac{\vDash M_i \in \text{Meth} \quad \iota_i \text{ distinct} \quad \forall\, i \in 1..n}{\iota_i\mapsto M_i\ ^{i\in 1..n} \vDash \diamond}$$

Result typing judgment: $\Sigma \vDash v : A$

(Result Object)

$$\frac{\Sigma \vDash \diamond \quad \Sigma_1(\iota_i) \equiv [l_i:\Sigma_2(\iota_i)\ ^{i\in 1..n}] \quad \forall\, i \in 1..n}{\Sigma \vDash [l_i=\iota_i\ ^{i\in 1..n}] : [l_i:\Sigma_2(\iota_i)\ ^{i\in 1..n}]}$$

Stack typing judgment: $\Sigma \vDash \mathcal{S} : E$

(Stack ø Typing)

$$\frac{\Sigma \vDash \diamond}{\Sigma \vDash \emptyset : \emptyset}$$

(Stack x Typing)

$$\frac{\Sigma \vDash \mathcal{S} : E \quad \Sigma \vDash [l_i=\iota_i\ ^{i\in 1..n}] : A \quad x \notin \text{dom}(E)}{\Sigma \vDash \mathcal{S}, x\mapsto[l_i=\iota_i\ ^{i\in 1..n}] : E, x:A}$$

Store typing judgment: $\Sigma \vDash \sigma$

(Store Typing)

$$\frac{\Sigma \vDash \mathcal{S}_i : E_i \quad E_i,\ x_i:\Sigma_1(\iota_i) \vdash b_i : \Sigma_2(\iota_i) \quad \forall\, i \in 1..n}{\Sigma \vDash \iota_i\mapsto\langle\varsigma(x_i)b_i, \mathcal{S}_i\rangle\ ^{i\in 1..n}}$$

We say that Σ' is an extension of Σ (and write $\Sigma' \geqslant \Sigma$) iff $\text{dom}(\Sigma) \subseteq \text{dom}(\Sigma')$ and for all $\iota\in\text{dom}(\Sigma)$, $\Sigma'(\iota) = \Sigma(\iota)$.

Lemma 4-1

If $\Sigma \vDash \mathcal{S} : E$ and $\Sigma' \vDash \diamond$ with $\Sigma' \geqslant \Sigma$, then $\Sigma' \vDash \mathcal{S} : E$.

□

If a term has a type, and the term reduces to a result in a store, then the result can be assigned that type in that store:

Soundness Theorem

If $E \vdash a : A \wedge \sigma\bullet\mathcal{S} \vdash a \rightsquigarrow v\bullet\sigma^\dagger \wedge \Sigma \vDash \sigma \wedge \text{dom}(\sigma)=\text{dom}(\Sigma) \wedge \Sigma \vDash \mathcal{S} : E$
then there exist a type $A^\dagger$ and a store type $\Sigma^\dagger$ such that:
$\Sigma^\dagger \geqslant \Sigma \wedge \Sigma^\dagger \vDash \sigma^\dagger \wedge \text{dom}(\sigma^\dagger)=\text{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \vDash v : A^\dagger \wedge A^\dagger <: A$.

Proof

By induction on the derivation of $\sigma\bullet\mathcal{S} \vdash a \rightsquigarrow v\bullet\sigma^\dagger$.

Case (Red x)

$$\frac{\sigma \bullet \mathcal{S}', x \mapsto [l_i = \iota_i \ ^{i \in 1..n}], \mathcal{S}'' \vdash \diamond}{\sigma \bullet \mathcal{S}', x \mapsto [l_i = \iota_i \ ^{i \in 1..n}], \mathcal{S}'' \vdash x \rightsquigarrow [l_i = \iota_i \ ^{i \in 1..n}] \bullet \sigma}$$

By hypothesis $E \vdash x : A \wedge \Sigma \vDash \sigma \wedge \mathrm{dom}(\sigma)=\mathrm{dom}(\Sigma) \wedge \Sigma \vDash \mathcal{S}', x \mapsto [l_i = \iota_i \ ^{i \in 1..n}], \mathcal{S}'' : E$. Because of $E \vdash x : A$, we must have $E \equiv E', x:A^\dagger, E''$ for some $A^\dagger <: A$.

Now, $\Sigma \vDash \mathcal{S}', x \mapsto [l_i = \iota_i \ ^{i \in 1..n}], \mathcal{S}'' : E$ must have been derived via several applications of (Stack x Typing) from, among others, $\Sigma \vDash [l_i = \iota_i \ ^{i \in 1..n}] : A^\dagger$.

Take $\Sigma^\dagger \equiv \Sigma$. We conclude $\Sigma^\dagger \vDash \sigma \wedge \mathrm{dom}(\sigma)=\mathrm{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \vDash [l_i = \iota_i \ ^{i \in 1..n}] : A^\dagger \wedge A^\dagger <: A$.

Case (Red Object)

$$\frac{\sigma \bullet \mathcal{S} \vdash \diamond \qquad \iota_i \notin \mathrm{dom}(\sigma) \qquad \iota_i \text{ distinct} \qquad \forall i \in 1..n}{\sigma \bullet \mathcal{S} \vdash [l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}] \rightsquigarrow [l_i = \iota_i \ ^{i \in 1..n}] \bullet (\sigma, \iota_i \mapsto \langle \varsigma(x_i) b_i, \mathcal{S} \rangle \ ^{i \in 1..n})}.$$

By hypothesis $E \vdash [l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}] : A \wedge \Sigma \vDash \sigma \wedge \mathrm{dom}(\sigma)=\mathrm{dom}(\Sigma) \wedge \Sigma \vDash \mathcal{S} : E$. Because of $E \vdash [l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}] : A$, we must have $E \vdash [l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}] : [l_i : B_i \ ^{i \in 1..n}]$ by (Val Object), for some $[l_i : B_i \ ^{i \in 1..n}] <: A$. Take $A^\dagger \equiv [l_i : B_i \ ^{i \in 1..n}]$.

Take $\Sigma^\dagger \equiv \Sigma, \iota_j \mapsto (A^\dagger \Rightarrow B_j) \ ^{j \in 1..n}$; by (Store Type) we have $\Sigma^\dagger \vDash \diamond$, because the $\iota_j \notin \mathrm{dom}(\sigma)$, and hence $\iota_j \notin \mathrm{dom}(\Sigma)$, and because $\vDash A^\dagger \Rightarrow B_j \in \mathrm{Meth}$ for $j \in 1..n$.

(1) Since $\Sigma^\dagger$ is an extension of Σ, by Lemma 4-1 we also have $\Sigma^\dagger \vDash \mathcal{S} : E$. Since $E \vdash [l_i = \varsigma(x_i) b_i \ ^{i \in 1..n}] : A^\dagger$, we must have $E, x_i : A^\dagger \vdash b_i : B_i$, that is, $E, x_i : \Sigma^\dagger{}_1(\iota_i) \vdash b_i : \Sigma^\dagger{}_2(\iota_i)$.

(2) We have that σ has the shape $\varepsilon_k \mapsto \langle \varsigma(x_k) b_k, \mathcal{S}_k \rangle \ ^{k \in 1..m}$. Now, $\Sigma \vDash \sigma$ must come from the (Store Typing) rule, with $\Sigma \vDash \mathcal{S}_k : E_k$ and $E_k, x_k : \Sigma_1(\varepsilon_k) \vdash b_k : \Sigma_2(\varepsilon_k)$. By Lemma 4-1, $\Sigma^\dagger \vDash \mathcal{S}_k : E_k$; moreover $E_k, x_k : \Sigma^\dagger{}_1(\varepsilon_k) \vdash b_k : \Sigma^\dagger{}_2(\varepsilon_k)$, because $\Sigma^\dagger(\varepsilon_k)=\Sigma(\varepsilon_k)$ for $k \in 1..m$ since $\mathrm{dom}(\sigma) = \mathrm{dom}(\Sigma) = \{\varepsilon_k \ ^{k \in 1..m}\}$ and $\Sigma^\dagger$ extends Σ.

By (1) and (2), via the (Store Typing) rule, we have $\Sigma^\dagger \vDash (\sigma, \iota_i \mapsto \langle \varsigma(x_i) b_i, \mathcal{S} \rangle \ ^{i \in 1..n})$. Since $\Sigma^\dagger \vDash \diamond$ and $\Sigma^\dagger \equiv \Sigma, \iota_j \mapsto (A^\dagger \Rightarrow B_j) \ ^{j \in 1..n}$, by the (Result Object) rule, we have $\Sigma^\dagger \vDash [l_i = \iota_i \ ^{i \in 1..n}] : A^\dagger$.

We conclude that $\Sigma^\dagger \geqslant \Sigma \wedge \Sigma^\dagger \vDash (\sigma, \iota_i \mapsto \langle \varsigma(x_i) b_i, \mathcal{S} \rangle \ ^{i \in 1..n}) \wedge \mathrm{dom}(\sigma, \iota_i \mapsto \langle \varsigma(x_i) b_i, \mathcal{S} \rangle \ ^{i \in 1..n})=\mathrm{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \vDash [l_i = \iota_i \ ^{i \in 1..n}] : A^\dagger \wedge A^\dagger <: A$.

Case (Red Select)

$$\frac{\sigma \bullet \mathcal{S} \vdash a \rightsquigarrow [l_i = \iota_i \ ^{i \in 1..n}] \bullet \sigma' \qquad \sigma'(\iota_j) = \langle \varsigma(x_j) b_j, \mathcal{S}' \rangle \qquad x_j \notin \mathrm{dom}(\mathcal{S}') \qquad j \in 1..n \qquad \sigma' \bullet \mathcal{S}', x_j \mapsto [l_i = \iota_i \ ^{i \in 1..n}] \vdash b_j \rightsquigarrow v \bullet \sigma''}{\sigma \bullet \mathcal{S} \vdash a.l_j \rightsquigarrow v \bullet \sigma''}.$$

By hypothesis $E \vdash a.l_j : A \wedge \Sigma \vDash \sigma \wedge \mathrm{dom}(\sigma)=\mathrm{dom}(\Sigma) \wedge \Sigma \vDash \mathcal{S} : E$. Since $E \vdash a.l_j : A$, we must have $E \vdash a : [l_j : B_j \ ...]$, for some $[l_j : B_j \ ...]$ with $B_j <: A$.

By the first induction hypothesis:

Since $E \vdash a : [l_j : B_j \ ...] \wedge \sigma \bullet \mathcal{S} \vdash a \rightsquigarrow [l_i = \iota_i \ ^{i \in 1..n}] \bullet \sigma' \wedge \Sigma \vDash \sigma \wedge \mathrm{dom}(\sigma)=\mathrm{dom}(\Sigma) \wedge \Sigma \vDash \mathcal{S} : E$, there exist a type A' and a store type Σ' such that:
$\Sigma' \geqslant \Sigma \wedge \Sigma' \vDash \sigma' \wedge \mathrm{dom}(\sigma')=\mathrm{dom}(\Sigma') \wedge \Sigma' \vDash [l_i = \iota_i \ ^{i \in 1..n}] : A' \wedge A' <: [l_j : B_j \ ...]$.

Since $\sigma'(\iota_j) = \langle \varsigma(x_j) b_j, \mathcal{S}' \rangle$, the judgment $\Sigma' \vDash \sigma'$ must come via (Store Typing) from $\Sigma' \vDash \mathcal{S}' : E_j$ and $E_j, x_j : \Sigma'{}_1(\iota_j) \vdash b_j : \Sigma'{}_2(\iota_j)$ for some E_j. Since $\Sigma' \vDash [l_i = \iota_i \ ^{i \in 1..n}] : A'$ must come from (Result Object), we have $A' \equiv [l_i : \Sigma'{}_2(\iota_j) \ ^{i \in 1..n}] \equiv \Sigma'{}_1(\iota_j)$. Since $A' <: [l_j : B_j \ ...]$, we have $\Sigma'{}_2(\iota_j) \equiv B_j$. Then, from $E_j, x_j : \Sigma'{}_1(\iota_j) \vdash b_j : \Sigma'{}_2(\iota_j)$ we obtain $E_j, x_j : A' \vdash b_j : B_j$. Moreover, by the (Stack x Typing) rule we get $\Sigma' \vDash \mathcal{S}', x_j \mapsto [l_i = \iota_i \ ^{i \in 1..n}] : E_j, x : A'$.

Let $E' \equiv E_j, x : A'$. By the second induction hypothesis:

Since $E' \vdash b_j : B_j \wedge \sigma' \bullet \mathcal{S}', x_j \mapsto [l_i = \iota_i \ ^{i \in 1..n}] \vdash b_j \rightsquigarrow v \bullet \sigma'' \wedge \Sigma' \vDash \sigma' \wedge \mathrm{dom}(\sigma')=\mathrm{dom}(\Sigma')$

$\wedge\ \Sigma' \vDash \mathcal{S}$, $x_j \mapsto [l_i = \iota_i\ ^{i \in 1..n}] : E'$,
there exist $A^\dagger$ and $\Sigma^\dagger$ such that:
$\Sigma^\dagger \geqslant \Sigma' \wedge \Sigma^\dagger \vDash \sigma'' \wedge \mathrm{dom}(\sigma'') = \mathrm{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \vDash v : A^\dagger \wedge A^\dagger <: B_j$.

We conclude:

- $\Sigma^\dagger \geqslant \Sigma$ by transitivity from $\Sigma^\dagger \geqslant \Sigma'$ and $\Sigma' \geqslant \Sigma$,
- $\Sigma^\dagger \vDash \sigma''$ with $\mathrm{dom}(\sigma'') = \mathrm{dom}(\Sigma^\dagger)$,
- $\Sigma^\dagger \vDash v : A^\dagger$ with $A^\dagger <: A$, by transitivity from $A^\dagger <: B_j$ and $B_j <: A$.

Case (Red Update)

$$\frac{\sigma \bullet \mathcal{S} \vdash a \rightsquigarrow [l_i = \iota_i\ ^{i \in 1..n}] \bullet \sigma' \qquad j \in 1..n \qquad \iota_j \in \mathrm{dom}(\sigma')}{\sigma \bullet \mathcal{S} \vdash a.l_j \Leftarrow \varsigma(x)b \rightsquigarrow [l_i = \iota_i\ ^{i \in 1..n}] \bullet \sigma'.\iota_j \leftarrowtail \langle \varsigma(x)b, \mathcal{S} \rangle}$$

By hypothesis $E \vdash a.l_j \Leftarrow \varsigma(x)b : A \wedge \Sigma \vDash \sigma \wedge \mathrm{dom}(\sigma) = \mathrm{dom}(\Sigma) \wedge \Sigma \vDash \mathcal{S} : E$.

Since $E \vdash a.l_j \Leftarrow \varsigma(x)b : A$, we must have $E \vdash a : [l_j : B_j \ldots]$ and $E, x : [l_j : B_j \ldots] \vdash b : B_j$ for some $[l_j : B_j \ldots] <: A$.

By induction hypothesis:

Since $E \vdash a : [l_j : B_j \ldots] \wedge \sigma \bullet \mathcal{S} \vdash a \rightsquigarrow [l_i = \iota_i\ ^{i \in 1..n}] \bullet \sigma' \wedge \Sigma \vDash \sigma \wedge \mathrm{dom}(\sigma) = \mathrm{dom}(\Sigma) \wedge \Sigma \vDash \mathcal{S} : E$,
there exist a type $A^\dagger$ and a store type $\Sigma^\dagger$ such that:
$\Sigma^\dagger \geqslant \Sigma \wedge \Sigma^\dagger \vDash \sigma' \wedge \mathrm{dom}(\sigma') = \mathrm{dom}(\Sigma^\dagger) \wedge \Sigma^\dagger \vDash [l_i = \iota_i\ ^{i \in 1..n}] : A^\dagger \wedge A^\dagger <: [l_j : B_j \ldots]$.

By assumption $\iota_j \in \mathrm{dom}(\sigma')$, hence $\iota_j \in \mathrm{dom}(\Sigma^\dagger)$. But $\Sigma^\dagger \vDash [l_i = \iota_i\ ^{i \in 1..n}] : A^\dagger$ must have been derived via (Result Object) from $\Sigma^\dagger{}_1(\iota_i) \equiv [l_i : \Sigma^\dagger{}_2(\iota_i)\ ^{i \in 1..n}] \equiv A^\dagger$ for all $i \in 1..n$. Hence, since $A^\dagger <: [l_j : B_j \ldots]$, we have $\Sigma^\dagger{}_2(\iota_j) = B_j$. Take $\sigma^\dagger \equiv \sigma'.\iota_j \leftarrowtail \langle \varsigma(x)b, \mathcal{S} \rangle$.

(1) We have $\Sigma^\dagger \vDash \mathcal{S} : E$ by Lemma 4-1. We also have $E, x : A^\dagger \vdash b : B_j$ by a bound change lemma (from $E, x : [l_j : B_j \ldots] \vdash b : B_j$ and $A^\dagger <: [l_j : B_j \ldots]$), that is $E, x : \Sigma^\dagger{}_1(\iota_j) \vdash b : \Sigma^\dagger{}_2(\iota_j)$.

(2) Since $\Sigma^\dagger \vDash \sigma'$ must come from (Store Typing), σ' has the shape $\varepsilon_k \mapsto \langle \varsigma(x_k)b_k, \mathcal{S}_k \rangle\ ^{k \in 1..m}$, and for all k such that $\varepsilon_k \neq \iota_j$ and for some E_k we have $\Sigma^\dagger \vDash \mathcal{S}_k : E_k$, and $E_k, x_k : \Sigma^\dagger{}_1(\varepsilon_k) \vdash b_k : \Sigma^\dagger{}_2(\varepsilon_k)$.

Then by (1) and (2) we have $\Sigma^\dagger \vDash \sigma'.\iota_j \leftarrowtail \langle \varsigma(x)b, \mathcal{S} \rangle$, by the (Store Typing) rule.

We conclude $\Sigma^\dagger \geqslant \Sigma \wedge \Sigma^\dagger \vDash \sigma^\dagger \wedge \Sigma^\dagger \vDash [l_i = \iota_i\ ^{i \in 1..n}] : A^\dagger$ and $A^\dagger <: A$ by transitivity from $A^\dagger <: [l_j : B_j \ldots]$ and $[l_j : B_j \ldots] <: A$.

Case (Red Clone)

$$\frac{\sigma \bullet \mathcal{S} \vdash a \rightsquigarrow [l_i = \iota_i\ ^{i \in 1..n}] \bullet \sigma' \qquad \iota_i \in \mathrm{dom}(\sigma') \quad \iota'_i \notin \mathrm{dom}(\sigma') \qquad \iota'_i \text{ distinct} \qquad \forall i \in 1..n}{\sigma \bullet \mathcal{S} \vdash \mathrm{clone}(a) \rightsquigarrow [l_i = \iota'_i\ ^{i \in 1..n}] \bullet (\sigma', \iota'_i \mapsto \sigma'(\iota_i)\ ^{i \in 1..n})}$$

By hypothesis $E \vdash \mathrm{clone}(a) : A \wedge \Sigma \vDash \sigma \wedge \mathrm{dom}(\sigma) = \mathrm{dom}(\Sigma) \wedge \Sigma \vDash \mathcal{S} : E$. Since $E \vdash \mathrm{clone}(a) : A$, we must have $E \vdash a : A$ (possibly via subsumption).

By the induction hypothesis:

Since $E \vdash a : A \wedge \sigma \bullet \mathcal{S} \vdash a \rightsquigarrow [l_i = \iota_i\ ^{i \in 1..n}] \bullet \sigma' \wedge \Sigma \vDash \sigma \wedge \mathrm{dom}(\sigma) = \mathrm{dom}(\Sigma) \wedge \Sigma \vDash \mathcal{S} : E$,
there exist a type $A^\dagger$ and a store type Σ' such that:
$\Sigma' \geqslant \Sigma \wedge \Sigma' \vDash \sigma' \wedge \mathrm{dom}(\sigma') = \mathrm{dom}(\Sigma') \wedge \Sigma' \vDash [l_i = \iota_i\ ^{i \in 1..n}] : A^\dagger \wedge A^\dagger <: A$.

Let $\Sigma^\dagger \equiv (\Sigma', \iota'_i \mapsto \Sigma'(\iota_i)\ ^{i \in 1..n})$ and $\sigma^\dagger \equiv (\sigma', \iota'_i \mapsto \sigma'(\iota_i)\ ^{i \in 1..n})$. We have $\Sigma^\dagger \vDash \diamond$ by (Store Type) because $\iota'_i \notin \mathrm{dom}(\sigma') = \mathrm{dom}(\Sigma')$, ι'_i are all distinct, and $\Sigma' \vDash \diamond$ is a prerequisite of $\Sigma' \vDash \sigma'$.

We conclude:

- $A^\dagger <: A$.
- $\Sigma^\dagger \geqslant \Sigma$, because $\Sigma' \geqslant \Sigma$ and $\Sigma^\dagger \geqslant \Sigma'$.
- $\mathrm{dom}(\sigma^\dagger) = \mathrm{dom}(\Sigma^\dagger)$, by construction and $\mathrm{dom}(\sigma') = \mathrm{dom}(\Sigma')$.

- $\Sigma^\dagger \vDash \sigma^\dagger$. Since $\Sigma' \vDash \sigma'$ must come from (Store Typing), σ' has the shape $\varepsilon_k \mapsto \langle \varsigma(x_k)b_k, \mathcal{S}_k \rangle^{\ k\in 1..m}$, and for all $k\in 1..m$ and for some E_k we have $\Sigma' \vDash \mathcal{S}_k : E_k$ and $E_k, x_k{:}\Sigma'_1(\varepsilon_k) \vdash b_k : \Sigma'_2(\varepsilon_k)$. Then also $E_k, x_k{:}\Sigma^\dagger_1(\varepsilon_k) \vdash b_k : \Sigma^\dagger_2(\varepsilon_k)$, and by Lemma 4-1 $\Sigma^\dagger \vDash \mathcal{S}_k : E_k$. Let $f{:}1..n \to 1..m$ be $\varepsilon^{-1}\cdot\iota$, so that for all $i\in 1..n$, $\iota_i = \varepsilon_{f(i)}$. We have $E_{f(i)}, x_{f(i)}{:}\Sigma'_1(\varepsilon_{f(i)}) \vdash b_{f(i)} : \Sigma'_2(\varepsilon_{f(i)})$ for $i\in 1..n$, so $E_{f(i)}, x_{f(i)}{:}\Sigma'_1(\iota_i) \vdash b_{f(i)} : \Sigma'_2(\iota_i)$. Moreover, since $\Sigma'(\iota_i) = \Sigma^\dagger(\iota'_i)$, we have $E_{f(i)}, x_{f(i)}{:}\Sigma^\dagger_1(\iota'_i) \vdash b_{f(i)} : \Sigma^\dagger_2(\iota'_i)$. The result follows by (Store Typing) from $\Sigma^\dagger \vDash \mathcal{S}_k : E_k$, and $\Sigma^\dagger \vDash \mathcal{S}_{f(i)} : E_{f(i)}$, and $E_k, x_k{:}\Sigma^\dagger_1(\varepsilon_k) \vdash b_k : \Sigma^\dagger_2(\varepsilon_k)$ and $E_{f(i)}, x_{f(i)}{:}\Sigma^\dagger_1(\iota'_i) \vdash b_{f(i)} : \Sigma^\dagger_2(\iota'_i)$, for $k\in 1..m$ and $i\in 1..n$.
- $\Sigma^\dagger \vDash [l_i = \iota'_i\ ^{i\in 1..n}] : A^\dagger$. First, $\Sigma' \vDash [l_i = \iota_i\ ^{i\in 1..n}] : A^\dagger$ must come from the (Result Object) rule with $A^\dagger \equiv \Sigma'_1(\iota_i) \equiv [l_i{:}\Sigma'_2(\iota_i)\ ^{i\in 1..n}]$ for $i\in 1..n$, and $\Sigma' \vDash \diamond$. But $\Sigma^\dagger(\iota'_i) \equiv \Sigma'(\iota_i)$ for $i\in 1..n$. So, $\Sigma^\dagger_1(\iota'_i) \equiv [l_i{:}\Sigma^\dagger_2(\iota'_i)\ ^{i\in 1..n}] \equiv A^\dagger$, and by (Result Object) $\Sigma^\dagger \vDash [l_i = \iota'_i\ ^{i\in 1..n}] : [l_i{:}\Sigma^\dagger_2(\iota'_i)\ ^{i\in 1..n}]$.

Case (Red Let)

$$\frac{\sigma\cdot\mathcal{S} \vdash b \rightsquigarrow v'\cdot\sigma' \qquad \sigma'\cdot\mathcal{S},\ x\mapsto v' \vdash c \rightsquigarrow v''\cdot\sigma''}{\sigma\cdot\mathcal{S} \vdash \text{let } x=c \text{ in } b \rightsquigarrow v''\cdot\sigma''}$$

By hypothesis $E \vdash \text{let } x=c \text{ in } b : A \ \wedge\ \Sigma \vDash \sigma \ \wedge\ \text{dom}(\sigma)=\text{dom}(\Sigma) \ \wedge\ \Sigma \vDash \mathcal{S} : E$. Since $E \vdash \text{let } x=c \text{ in } b : A$, we must have $E \vdash c : C$ for some C, and $E, x{:}C \vdash b : A$ (possibly via subsumption).

By the first induction hypothesis:

Since $E \vdash c : C \ \wedge\ \sigma\cdot\mathcal{S} \vdash c \rightsquigarrow v'\cdot\sigma' \ \wedge\ \Sigma \vDash \sigma \ \wedge\ \text{dom}(\sigma)=\text{dom}(\Sigma) \ \wedge\ \Sigma \vDash \mathcal{S} : E$,
there exist a type C' and a store type Σ' such that:
$\Sigma' \geqslant \Sigma \ \wedge\ \Sigma' \vDash \sigma' \ \wedge\ \text{dom}(\sigma')=\text{dom}(\Sigma') \ \wedge\ \Sigma' \vDash v' : C' \ \wedge\ C' <: C$.

By Lemma 4-1, $\Sigma' \vDash \mathcal{S} : E$, hence by (Stack x Typing) $\Sigma' \vDash \mathcal{S},\ x\mapsto v' : E, x{:}C'$. From $E, x{:}C \vdash b : A$ by bound weakening, $E, x{:}C' \vdash b : A$.

By the second induction hypothesis:

Since $E, x{:}C' \vdash b : A \ \wedge\ \sigma'\cdot\mathcal{S},\ x\mapsto v' \vdash b \rightsquigarrow v''\cdot\sigma'' \ \wedge\ \Sigma' \vDash \sigma' \ \wedge\ \text{dom}(\sigma')=\text{dom}(\Sigma')$
$\wedge\ \Sigma' \vDash \mathcal{S},\ x\mapsto v' : E, x{:}C'$,
there exist a type $A^\dagger$ and a store type $\Sigma^\dagger$ such that:
$\Sigma^\dagger \geqslant \Sigma' \ \wedge\ \Sigma^\dagger \vDash \sigma'' \ \wedge\ \text{dom}(\sigma'')=\text{dom}(\Sigma^\dagger) \ \wedge\ \Sigma^\dagger \vDash v'' : A^\dagger \ \wedge\ A^\dagger <: A$.

We conclude that $\Sigma^\dagger \geqslant \Sigma$ (by transitivity), $\Sigma^\dagger \vDash \sigma''$ with $\text{dom}(\sigma'')=\text{dom}(\Sigma^\dagger)$, and $\Sigma^\dagger \vDash v'' : A^\dagger$ with $A^\dagger <: A$.

□

Corollary

If $\emptyset \vdash a : A$ and $\emptyset\cdot\emptyset \vdash a \rightsquigarrow v\cdot\sigma$
then there exist a type $A^\dagger$ and a store type $\Sigma^\dagger$ such that
$\Sigma^\dagger \vDash \sigma$ and $\Sigma^\dagger \vDash v : A^\dagger$, with $A^\dagger <: A$.

□

Therefore, if a term has a type, and the term reduces to a result in a store, then the result can be assigned that type in that store. That is, if a term produces a result, it does so by respecting the type that it had been assigned statically.

This statement is vacuous if the term does not produce a result. This can happen either because reduction diverges (the rules are applicable ad infinitum), or because it gets stuck (no rule is applicable at a certain stage).

For each term there is at most one rule whose conclusion matches the syntactic form of the term, and hence is potentially applicable to the term. The rule (Red x) is applicable to x unless x is not defined in the stack. Assuming that b reduces to v, the rule (Red Update) is applicable to $b.l_j \Leftarrow \varsigma(x)c$ provided v has l_j. The applicability of the rule (Red Select) is determined with an analo-

gous condition. Assuming the appropriate subterms converge, the rules (Red Object), (Red Clone), and (Red Let) are always applicable to terms of the corresponding forms. Examining these cases, we can prove that the reduction of a well-typed term in a well-typed store cannot get stuck (although it may diverge).

5. Conclusions

We view our calculus as a small kernel for object-oriented languages. (In fact, its primitives have been used in the Obliq distributed scripting language [11].) The calculus is not class-based, since classes are not built-in, nor delegation-based [25] since the method-lookup mechanism does not delegate invocations. However, the calculus models class-based languages well, as we show in [4, 5]. In delegation-based languages, traits play the role of classes. Our calculus can model traits just as easily as classes, along with dynamic inheritance based on traits. Interpreting delegation fully, though, would require significant formal complications, because of the complexity of method lookup in delegation.

References

[1] Abadi, M., **Baby Modula-3 and a theory of objects**. *Journal of Functional Programming* **4**(2), 249-283. 1994.

[2] Abadi, M. and L. Cardelli, **A semantics of object types**. *Proc. IEEE Symposium on Logic in Computer Science*. 1994.

[3] Abadi, M. and L. Cardelli, **A theory of primitive objects: second-order systems**. *Proc. ESOP'94 - European Symposium on Programming*. Springer-Verlag. 1994.

[4] Abadi, M. and L. Cardelli, **A theory of primitive objects: untyped and first-order systems**. *Proc. Theoretical Aspects of Computer Software*. Springer-Verlag. 1994.

[5] Abadi, M. and L. Cardelli, **An imperative object calculus**. *Proc. TAPSOFT'95 (to appear)*. Springer-Verlag. 1995.

[6] Birtwistle, G.M., O.-J. Dahl, B. Myhrhaug, and K. Nygaard, **Simula Begin**. Studentlitteratur. 1979.

[7] Borning, A.H., **Classes versus prototypes in object-oriented languages**. *Proc. ACM/IEEE Fall Joint Computer Conference*. 1986.

[8] Bruce, K., **A paradigmatic object-oriented programming language: design, static typing and semantics**. *Journal of Functional Programming* **4**(2), 127-206. 1994.

[9] Bruce, K. and R. van Gent, **TOIL: A new type-safe object-oriented imperative language**. Manuscript. 1993.

[10] Cardelli, L., **Extensible records in a pure calculus of subtyping**. In *Theoretical Aspects of Object-Oriented Programming*, C.A. Gunter and J.C. Mitchell, ed. MIT Press. 373-425. 1994.

[11] Cardelli, L., **Obliq: A language with distributed scope**. Report n.122. Digital Equipment Corporation, Systems Research Center. 1994.

[12] Cardelli, L. and J.C. Mitchell, **Operations on records**. *Mathematical Structures in Computer Science* **1**(1), 3-48. 1991.

[13] Chambers, C., D. Ungar, B.-W. Chang, and U. Hölzle, **Parents are shared parts of objects: inheritance and encapsulation in Self**. *Lisp and Symbolic Computation* **4**(3). 1991.

[14] Dony, C., J. Malenfant, and P. Cointe, **Prototype-based languages: from a new taxonomy to constructive proposals and their validation.** P*roc. OOPSLA'92*. 1992.

[15] Eifrig, J., S. Smith, V. Trifonov, and A. Zwarico, **An interpretation of typed OOP in a language with state**. Dept. of Computer Science, The Johns Hopkins University. 1993.

[16] Harper, R., **A simplified account of polymorphic references**. *Information Processing Letters* **51**(4). 1994.

[17] Harper, R. and B. Pierce, **A record calculus based on symmetric concatenation**. *Proc. 18th Annual ACM Symposium on Principles of Programming Languages*. 1991.

[18] Leroy, X., **Polymorphic typing of an algorithmic language**. Rapport de Recherche no.1778 (Ph.D Thesis). INRIA. 1992.

[19] Mitchell, J.C., F. Honsell, and K. Fisher, **A lambda calculus of objects and method specialization**. *Proc. 8th Annual IEEE Symposium on Logic in Computer Science*. 1993.

[20] Pierce, B.C. and D.N. Turner, **Simple type-theoretic foundations for object-oriented programming**. *Journal of Functional Programming* **4**(2), 207-247. 1994.

[21] Rémy, D., **Typechecking records and variants in a natural extension of ML**. *Proc. 16th Annual ACM Symposium on Principles of Programming Languages*. 1989.

[22] Stein, L.A., H. Lieberman, and D. Ungar, **A shared view of sharing: the treaty of Orlando**. In *Object-oriented concepts, applications, and databases,* W. Kim and F. Lochowsky, ed. Addison-Wesley. 31-48. 1988.

[23] Taivalsaari, A., **Object-oriented programming with modes**. *Journal of Object Oriented Programming* **6**(3), 25-32. 1993.

[24] Tofte, M., **Type inference for polymorphic references**. *Information and Computation* **89**, 1-34. 1990.

[25] Ungar, D. and R.B. Smith, **Self: the power of simplicity**. *Lisp and Symbolic Computation* **4**(3). 1991.

[26] Wand, M., **Type inference for record concatenation and multiple inheritance**. *Proc. 4th Annual IEEE Symposium on Logic in Computer Science*. 1989.

[27] Wright, A.K. and M. Felleisen, **A syntactic approach to type soundness**. *Information and Computation* **115**(1), 38-94. 1994.

[28] Yonezawa, A. and M. Tokoro, ed. **Object-oriented concurrent programming**. MIT Press. 1987.

Lazy Computations in an Object-Oriented Language for Reactive Programming

Johan Nordlander
Department of Computing Science
Chalmers University of Technology
S - 412 96 Göteborg, Sweden
Email: nordland@cs.chalmers.se

December 22, 1994

Abstract

We present a model of programming which includes lazy expression evaluation as well as communication between parallel objects encapsulating a local state. The purpose of introducing objects in this model is to make explicit the context in which functional programs execute, in order to separate the abstract results a system computes from the interaction it maintains with its environment. We use the word *reactive* to describe this model, which emphasizes our view that the utmost purpose of a computer system is to react to stimuli from the outside.

The reactive programming model consists of a combination of object-oriented and functional styles, in such a way that the object-oriented level of programming has access to the functional world, but not vice versa. By keeping this distinction, our expression sublanguage can maintain the typical properties of purely functional languages: non-strict semantics, freedom of side-effects, etc. In this presentation, we illustrate our notion of reactive programming by means of a programming language, for which we give an operational semantics, as well as a couple of examples of typical use.

1 Introduction

What is the goal for introducing a notion of state in functional languages? Are we primarily interested in making our languages more adapted to the machine or to the programmer?

Programming with state is generally regarded as a rather machine-oriented way of programming. This is especially true when we compare with functional languages, which support a high level of abstraction and are designed without any specific machine-architecture in mind. Thus, the motivation for introducing state in a functional language is mostly pragmatic: it gives the programmer an opportunity to write more efficient code than what might otherwise be produced by the compiler. The challenge to the language designer is then how to encapsulate such pieces of stateful code so that they may be treated as genuine functions when called from the outside. In these settings, stateful programming is considered an alternative, but obviously *less* human-friendly means of expressing something which is still preferably thought of as a function.

However, we consider the notion of state to be equally important when it comes to making functional languages *more* human-friendly. This is not a manifestation of some idea that imperative algorithms should be easier to grasp for beginners — in fact we think that the opposite situation is more likely to be true. We advocate a model of state because of a perceived limitation of the

functional programming paradigm: its inherent intuition that the computer is a programmable calculator, whose purpose is to deliver a result when fed with a function and some input data. However true this view may have been in the dawn of the computer era, users of modern personal computers probably view their machines more as flexible storage-devices, databases, simulators (of office desktops and cockpits), controllers (of radiators and telephone lines), etc; i.e. machines that *maintain an interaction* with their environment. In the description of such machines, we find it very unintuitive to focus on what *result* is being computed (what is really the result of an operating system?).

In our view, a far more statisfactory model concentrates on the *state* of a computer system — how it is composed and how it *reacts* to various events. The argument for this view is simply that humans (including programmers) tend to interpret their environment (including computers) in terms of state and change. Files, screens, windows, etc, are arguably more easily thought of as unique objects with a changeable state, than as the mere representation of some primary, infinite expression under evaluation. On the other hand, many computing tasks involve no interesting states apart from their results, nor do they need to synchronize with the external world. For such tasks, a state model becomes equally unpleasant, since humans generally prefer to understand these tasks as abstract mappings. So, what we would really like is a kind of hybrid model, which uses functions to describe mappings, and state to capture interaction.

In this paper, we present an attempt towards this goal, by means of a lazy functional language extended with concepts borrowed from object-oriented technology. As we have indicated above, we are not concerned with retaining the *programs-as-functions* paradigm of functional languages; instead we favour a *reactive* model, where a program is a dynamic collection of transiently active objects managing a local state. We are, however, very concerned with not destroying the main virtues of lazy functional languages, i.e. those virtues that permit unrestricted equational reasoning about expressions. Thus, in our model of programming, expression evaluation is free from side-effects, referentially transparent, and independent of order (assuming termination). This is achieved by an asymmetric combination of object-oriented and functional programming, such that the object-oriented level is aware of the functional world, but not vice versa. The vital properties of the reactive model are summarized below:

- It is *object-oriented*, in the sense that it supports and enforces partitioning of a system state into small, self-contained objects.
- Objects are *reactive*, by which we mean that an object will do nothing but maintaining its state until it receives an external request to execute some *action*. Actions may involve an arbitrary amount of computation and communication, but, in the absence of non-termination and deadlock, every object will eventually enter a resting state again. There are no iterative loops in this model.
- Objects are independent and may evolve in parallel. Both synchronous and asynchronous communication between objects is supported, and mutual exclusion between different actions of an object is guaranteed. Thus, we identify the notions of objects and processes.
- It uses a functional sublanguage with non-strict semantics to express computations. This sublanguage retains all the semantic properties of pure functional languages like Haskell [HJW+91]. In particular, any Haskell program not depending on a particular I/O model can easily be expressed.
- Expressions are only evaluated on demand by objects during action execution. There is no *print-loop* involved, and no particular *main* expression. A crucial property is that expression evaluation can neither cause nor detect any state change in the object world.

- Objects may be created dynamically at run-time by instantiating object *templates*, and the configuration of objects may be dynamically changed, by sending messages containing *object references*.
- Input is modelled as a request to execute some action of a certain object, while output is a state-change of some object belonging to the environment. This model extends naturally to include the treatment of input as interrupts and output as the state-change of hardware objects like the display memory.
- Actions, references and templates are first-class values, which means that they may be used as parameters, function results, data structure members, and state contents. This allows large systems of objects to be decomposed into smaller ones in a way very similar to hardware design.

It is important to note that this asymmetric two-level design does not permit the use of action execution in the encoding of functions. This means that we support imperative algorithmic programming no better than, say, Haskell. As we have mentioned, the purpose of our work is to make object-oriented, reactive I/O available to the functional programmer. However, the undistorted semantics of our functional sublanguage should enable any functionally consistent imperative extension to be directly applied, if desired. How this can be achieved without introducing different notations for similar concepts is an area of future work, though.

We will continue this paper by giving a brief overview of *Omelett*[1], a language designed to illustrate our notion of reactive programming. Then, in section 3, we will present some non-trivial programming examples using this language. In section 4 we will give the formal definition of the dynamic semantics of Omelett. The paper concludes in section 5 with a comparison to related work. The most obvious omissions from this presentation are the issues of static semantics and implementation techniques, matters which are fully covered in our licentiate thesis [Nor94].

2 Overview of Omelett

A central idea behind our approach is the distinction between a *computation*, which serves to deliver a result as fast as possible, and a *simulation*, which maintains a possibly infinite interaction with its environment. In this setting, a compiler performs a typical computation, while a text-editor is engaged in a simulation (of a type-writer in this case). A similar distinction can be made between the use of *values* and *objects* in programming languages. In our terminology, values stand for the constant, timeless abstractions we know from mathematics. This contrasts to objects, which are concrete models of the real world where terms like state and change are valid. Arguably, the *value-oriented* and *object-oriented* programming styles are ideally suited to the tasks of computation and simulation, respectively.

Omelett supports user-defined values as well as user-defined objects, by means of two distinct kinds of type-declarations. Having both notions at hand within the same language gives the programmer freedom in choosing the best representation at every stage: if something can be thought of as having a unique identity throughout potential state changes, it should be coded as an object; otherwise it should be coded as a data value.

We may illustrate this philosophy by a small example. Take the concrete activity of repainting a red car with blue color. In a functional style, this has to be interpreted as the definition of a new car value, distinct from the red one. Under the object-oriented paradigm we may retain the identity of

[1] *Object-oriented MEta Language ETT (1).*

the car object, but we run the risk of accidentally changing the meaning of object "red" instead. In the combined, reactive style of Omelett, we are able to code the example according to our intuition. By treating the car as an object and the colors as values, repainting is formulated as the action that changes the color-state of the car from the value of red to the value of blue. We run no risk of disrupting the meaning of any expression because of the state change, still we automatically capture the idea that the repainted car is *the same* as the old one.

The example looks as follows when coded in Omelett:

```
data Color is
    Red
    Green
    Blue

object Car is
    orig_color :: Color
    curr_color :: ! Color
    repaint    :: Color -> !!

template car init_color is
    orig_color = init_color
    action curr_color does
        reply color
    action repaint new_color does
        set color = new_color
    state color = init_color

new my_car <- car Red
```

Here `Car` is the name of a new type whose values are *references* to objects having the visible attributes `orig_color`, `curr_color`, and `repaint`. A template for such objects is obtained by applying `car` to a value of type `Color`. This is done in the `new` construct on the last line, which binds `my_car` to a fresh object reference of type `Car`. It should be emphasized that all three attributes are really constant values; for example, `curr_color` is not a value of type `Color`, but a *synchronous action* whose replies are of type `Color`. Such a reply may only be collected by some other action under execution, by writing

```
req c <- my_car.curr_color
```

Likewise, the color of `my_car` may only be changed by means of the asynchronous form of the same command:

```
req my_car.repaint Blue
```

These actions may be requested in parallel by multiple objects — the semantics of objects still guarantees that `my_car` will execute at most one of its actions at a time. On the functional level, though, it is only possible to *select* action values, not to request their execution. We may say that an action value is a reaction *capability*, that can only be utilized outside the language of expressions. This is the feature which makes expression evaluation free from side-effects.

The full syntax of Omelett is given in figure 1[2]. A few further remarks may help clarifying its intended semantics:

[2]In this grammar we take the liberty to let a postfix *s* denote a sequence of the non-terminal to which it is attached.

prog	⟶	*tdecls decls*	Top-level declarations
tdecl	⟶	`data` *tcon tvars* `is` *constrs*	Inductive datatype
	|	`object` *tcon tvars* `is` *selects*	Object reference type
constr	⟶	*con types*	Data constructor declaration
select	⟶	*sel* :: *type*	Attribute selector declaration
type	⟶	*tvar*	Type variable
	|	*tcon types*	Saturated type constructor
	|	*type* -> *type*	Function type
	|	* *type*	Template type
	|	! *type*	Synchronous action type
	|	!!	Asynchronous action type
decl	⟶	*bind*	Standard global declaration
	|	`new` *gens*	Global object declarations
	|	`template` *var vars* `is` *attrs* `state` *binds*	Template declaration
attr	⟶	*bind*	Standard attribute
	|	`new` *gens*	Sub-object attributes
	|	`action` *var vars* `does` *cmds*	Action attribute
bind	⟶	*var* = *expr*	Value binding
gen	⟶	*var* <- *expr*	Generator
expr	⟶	*var*	Variable
	|	*con*	Data constructor
	|	*var* -> *expr*	Abstraction
	|	*expr expr*	Application
	|	*expr*.*sel*	Attribute selection
	|	`let` *binds* `in` *expr*	Local bindings
	|	`case` *expr* `of` *ealts*	Case analysis
ealt	⟶	*con vars* -> *expr*	Expression alternative
cmd	⟶	`set` *binds*	Assignment
	|	`new` *gens*	Object creation
	|	`req` *exprs*	Asynchronous action request
	|	`req` *gens*	Synchronous rendezvous
	|	`reply` *expr*	Rendezvous reply
	|	`let` *binds*	Local bindings
	|	`case` *expr* `of` *calts*	Case analysis
calt	⟶	*con vars* -> *cmds*	Command alternative

Figure 1: Omelett syntax

- Only action commands may refer to the state variables of a template. This restriction particularly precludes state dependent attributes.
- All template declarations are automatically in the scope of the pre-declared variable `self`. This variable becomes bound to the actual object reference created during instantiation.
- The `new` construct appears as a global declaration, as an attribute, and as an action command, but *not* as an expression alternative. If the latter form were allowed, objects could be created as a side-effect of evaluation, and referential transparency would be lost.
- Assignments and object instantiations are performed lazily, just like local bindings. This makes the `case` and the `req` commands the only real forces behind expression evaluation.

3 Some example programs

In this section we will examine two simple, but nevertheless typical programming problems. The examples are of necessity incomplete, but we think that they still serve the purpose of illustrating some of the main features of reactive programming. In order to make the examples look less esoteric, we will use conventional syntax for numbers, strings, and lists, and we will also make some limited use of established functional language features like pattern bindings and list comprehensions.

3.1 Controller for an autonomously guided vehicle

Our first program illustrates the separation between the computations performed by a program, and the simulation in which it is involved. Since it is an example of an interrupt-driven system with parallel processes, which also performs heavy calculations, it captures many of the characteristics of a typical stand-alone, industrial system.

The concrete task is to control an autonomously guided vehicle (AGV). Such a vehicle is capable of navigating (within a limited area) without the guidance of a human driver. The crucial point for an AGV is to know its position at all times. There are different methods developed for this purpose, the one we will consider in our example relies on the existence of a set of *reflectors* placed out at known positions within the room in question [Hyp87]. To determine its position, the AGV measures angles to the visible reflectors by means of a rotating laser beam. The angles are then compared with the positions of the reflectors, to yield a position within the room from which the angles must have been measured. If this position does not coincide with the desired position at that time, a regulating algorithm generates appropriate adjustments to the driving and steering servos.

Our first step in programming a controller for such an AGV is to separate the two major algorithms — positioning and regulation — from the questions of hardware interaction. The algorithms, called `calcpos` and `regulate`, are of course coded as functions, but since their implementation mainly concerns matters that belong to the field of control theory, we only give their type signatures (using a syntax similar to the selector declarations). Figure 2 shows the resulting code.

Viewing the system from the outside we identify three main objects (or processes): the angle-meter, the simulated driver, and the servo. These objects are defined by giving their interfaces (the object types) as well as their implementation templates.

The AGV controller must obviously be a sampling system, and the clock driving this implementation is actually the rotating beam, which issues a zero-crossing interrupt (tick) at the completion of each turn. The interrupt is reacted to by the angle-meter action `zerocross`. Furthermore, during the scan of the beam, the meter hardware generates another interrupt at each detection of a reflector, which is handled by the action `detect` within the same process.

```
calcpos  :: [Angle] -> [Pos] -> Pos
regulate :: Pos -> Pos -> Speed -> Speed
room     :: [Pos]

object Driver is
    newscan   :: [Angle] -> !!
    newpath   :: [Pos] -> !!
object Meter is
    detect    :: !!
    zerocross :: !!
object Servo is
    setspeed  :: Speed -> !!

template driver s is
    action newscan angles does
        let current       = calcpos angles room
            desired:path' = path
            speed'        = regulate current desired speed
        req s.setspeed speed'
        set speed = speed'
            path  = path'
    action newpath p does
        set path = p
    state speed = zero
          path  = cycle origo

template meter d is
    new angle_reg <- in_port angle_reg_addr
    action detect does
        req a <- angle_reg.read
        set angles = a:angles
    action zerocross does
        req d.newscan angles
        set angles = []
    state angles = []

new d <- driver s
    m <- meter d
    s <- servo ...
```

Figure 2: A controller program for an autonomously guided vehicle

In the body of `detect` we touch the issue of how concrete hardware interaction can be performed. The angle-meter process has created a subobject `angle_reg` at startup, by instantiating a template `in_port` which we assume is primitive. Because of the address given to `in_port` at instantiation, `angle_reg` maps to the actual hardware register that contains the current angle of the beam, and this angle is read by `detect` and inserted in the list of angles collected during the present scan. When a `zerocross` interrupt is issued, the angle-meter process sends the accumulated list to the driver process, and clears its local state.

The state of the driver process consists of the current speed vector, and a list of positions that constitute the path that the AGV has to follow. The path list is consumed one element per system tick, and in order to make our example simple, we have assumed that the list is infinite. Thus, to make the AGV stop, the path must contain the same position repeatedly. The driver action `newpath` can be requested at any time to give the AGV a new path to follow. We are not concerned here with the possible origins of such requests, however.

At each tick, the angle-meter process sends the detections of the completed scan to the driver by requesting the action `newscan`. The body of this action has in fact only two concrete activities to perform: to update the servo with a new speed vector, and to advance the path state by one step. The remaining issues of `newscan` are preferably expressed in the functional domain, using the `let` construct at the beginning.

The servo process is only given as an interface. It may be implemented in hardware directly, or it may be a software object of any complexity. All that we need to know is that it accepts new speed vectors by means of an action named `setspeed`.

Finally, the global `new` declaration ties it all together, by instantiating processes (objects) from the templates. What we have not shown is how the interrupts of the system are directed to the two actions of the angle-meter process. Since this coupling actually forms the "top" of a stand-alone, reactive system, we could imagine a convention that requires the interrupt vector table to be declared in the program text under the name `main`.

3.2 Desktop calculator

The next example of a reactive program is a "desktop" calculator, a common application under modern operating systems that utilize the desktop metaphor as its user interface. Apart from illustrating how a typical interactive program can be coded in Omelett, it also shows an application of both higher-order functions and higher-order actions.

In this example we will rely on the existence of a graphical user-interface library, based on the notion of graphical objects of type `View`. A view is an object capable of drawing a representation of itself on the screen, and the library is assumed to contain templates for many different view objects like buttons, text displays, windows, menus, etc. Furthermore, the run-time interaction with the user is supposed to be administered by a *view manager* process, which takes care of tasks like reacting to mouse movements, redrawing the screen, and directing input events to the view pointed to by the mouse. Our job is to supply a template that defines the unique characteristics of a *calculator* view.

Figure 3 shows our implementation of the calculator template. The looks of the calculator are defined by instantiating a view structure starting at attribute `body`, utilizing primitive view templates (`button` and `display`) as well as designated *layout* templates (`matrix` and `vbox`) fetched from the library. In the declaration of the `digits` attribute, we see an example of how list comprehensions can be used to generate lists, in this case a list of button templates. Omelett lets the `new` construct (as well as the `req` commands) distribute over lists, so the result of the instantiation will be a list of button objects, bound to the name `digits`.

The `button` template is parameterized with respect to a button label and an asynchronous action

```
display :: String -> *View
button  :: String -> !! -> *View
matrix  :: Int -> Int -> [View] -> *View
vbox    :: [View] -> *View

template calculator is

    new displ  <- display "0"
        digits <- [button (show n) (do_digit n) | n <- [0..9]]
        ops    <- [button (show f) (do_op f) | f <- [(+),(-),(*),(/)]]
        pad    <- matrix 4 4 (button "ENT" do_enter :
                              button "CLR" do_clear :
                              digits ++ ops)
        body   <- vbox [displ,pad]

    action do_digit n does
        set fst = 10 * fst + n
        req displ.draw (show fst)

    action do_enter does
        set fst:rest = 0:fst:rest

    action do_op f does
        case rest of
            snd:rest' ->
                let result = f snd fst
                set fst:rest = result:rest'
                req displ.draw (show result)
            [] ->
                req env.beep

    action do_clear does
        req displ.draw "0"
        set fst:rest = [0]

    state fst:rest = [0]
```

Figure 3: A desktop calculator

that the button objects should request when hit by the mouse. The action argument is formed by applying the local action `do_digit` to the number of the button. It is important to observe that this application does not imply any *execution* of `do_digit` at this point. As we have stressed before, action values only denote the capability to react, not the reaction itself. The situation is more or less the same in the instantiation of the `ops` attribute; here the arguments to the local action `do_op` are *functions*. In order to avoid clutter, we have assumed that there exists a function `show` that returns a suitable string representation of functions (!) as well as numbers.

There are two more actions in the calculator template, `do_enter` and `do_clear`, which correspond to the "ENT" and "CLR" buttons of our calculator, respectively. Furthermore, the internal state of the calculator is a stack of numbers, defined using pattern matching to get separate names for the first element and the rest of the stack. When requested, that is when a graphical button is hit, `do_digit` will update the head of the stack and redraw the display. If the "ENT" button is pressed, a `0` is pushed on top of the stack, but the display is not redrawn in order not to confuse the user.

`do_op` requires at least two numbers on the stack, or else it will issue a "beep" request to some global object `env`. If there are enough numbers, the function parameter (which will be `(+)`, `(-)`, `(*)`, or `(/)`, depending on the button pressed) is applied to the two topmost elements, which are replaced with the result. Finally, the result is drawn on the display. The behaviour of the last action, `do_clear`, is obvious.

4 Operational semantics

In order to formally define the semantics of Omelett, we need a way of representing objects and references which is consistent with the representation of lazy evaluation, where sharing and updating must also be easy to express. Our approach is to build on work by Launchbury [Lau93], who defines an operational semantics for lazy evaluation by modelling the run-time heap as a set of recursive bindings, where the variables take the role of pointers. This of course requires all bound variables to be distinct, and we will simply assume that the Omelett source is accordingly renamed to begin with.

Before that, however, we need to perform some slight source transformations, so that we can express all global declarations and attribute bindings on the *var* = *expr* form. This requires the following addition to the syntax for expressions:

$$expr \longrightarrow \texttt{action}\ var\ cmds \mid \{\, cmds \mid vars \mid \sigma \,\}$$

Here the first expression form represents an action value (tied to a specific object via the *var* component), while the second form will constitute *object nodes* in the heap. Object nodes are introduced in the transformation of templates, which are turned into functions abstracting out the variable `self` from an object node. This scheme also makes object instantiation easy — it just becomes the binding of an object name (reference) v to the template expression applied to the same v. Figure 4 gives the complete transformation scheme.

An object node has three components: a current command sequence representing its "program counter", the roots of its attribute expressions, and its local state. The state is modelled as a *substitution*, which binds the state variables (free in the action bodies) to variables denoting the current state contents. We use σ to range over substitutions, and a standard postfix $[\sigma]$ notation for the renaming of a construct according to σ. Unique variable names are generated within the meta-function NEW(), defined by:

$$\textsc{new}(v_1 \ldots v_n) \equiv v_1 = v_1' \cdots v_n = v_n' \qquad \text{where } v_1' \ldots v_n' \text{ are fresh variables}$$

$$\texttt{action}\ v\ vs\ \texttt{does}\ cs \quad \rightsquigarrow \quad v = \backslash vs \to \texttt{action self}\ cs$$

$$\texttt{template}\ v\ vs\ \texttt{is}\ bs\ \texttt{state}\ bs' \quad \rightsquigarrow \quad v = \backslash vs \to \backslash \texttt{self} \to \texttt{let}\ bs +\!\!+ bs'[\sigma] \ \texttt{in}\ \{\_ \mid \textsc{dv}(bs) \mid \sigma\}$$

where $\sigma \equiv \textsc{new}(\textsc{dv}(bs'))$

$$\texttt{new}\ v_1 \leftarrow e_1 \cdots v_n \leftarrow e_n \quad \rightsquigarrow \quad \texttt{let}\ v_1 = (e_1\ v_1) \cdots v_n = (e_n\ v_n) \qquad \text{(commands)}$$

$$\texttt{new}\ v_1 \leftarrow e_1 \cdots v_n \leftarrow e_n \quad \rightsquigarrow \quad v_1 = (e_1\ v_1) \cdots v_n = (e_n\ v_n) \qquad \text{(declarations/attributes)}$$

Figure 4: Initial transformation of Omelett source

Our meta-notation furthermore lets $\_$, :, and $+\!\!+$ stand for the empty sequence, sequence construction, and sequence concatenation, respectively. We will also use $\oplus$ as an alternative to $+\!\!+$, for both forming and matching against sequences where the actual order is irrelevant. $\textsc{dv}(bs)$ finally denotes the variables defined in *bs*.

The operational semantics is defined as a transition system between abstract machine configurations. We actually make use of two transition relations, $\Longrightarrow$ and $\Downarrow$, where $\Longrightarrow$ defines command execution, and $\Downarrow$ defines expression evaluation. A configuration is just a heap (Γ) in the former case, and a pair of a heap and an expression ($\Gamma \triangleright e$) in the latter. The asymmetry in our combination of object- and value-oriented programming is evident in the semantics in the sense that $\Longrightarrow$ is defined in terms of $\Downarrow$, but not vice versa. The initial configuration of the system is the heap formed by the transformed and renamed source code, and the meaning we assign to our programs is the set of possible configuration sequences defined by $\Longrightarrow$. Figure 5 shows the full set of transition rules.

The semantics of expression evaluation (rules *a* to *i*) is straightforward, we particularly note that the two additions to the expression syntax form canonical values (rules *h* and *i*). The hat notation $\widehat{e}$ in rule *a* is Launchbury's syntax for dynamic renaming of bound variables in *e*. This renaming is necessary in order to keep the program distinctly named, since expressions referenced by name may be shared. Renaming is trivial and need not be defined here, we only have to note that the variables introduced in the two extra expression forms are not binding occurrences. Apart from the addition of objects and action values, the remaining difference between our formulation and Launchbury's semantics is that we put function arguments in the heap, as if they were local bindings. Launchbury instead does meta substitution of the argument in the function body, but this requires explicit naming of every argument in order to preserve laziness. Our approach is more straightforward, but it also means that we introduce an extra level of indirection for arguments which already are variables. However, since we are not primarily interested in the space behaviour of programs, this has no significance.

The rules defining command execution (*j* to *p*) focus on a specific object node, which must generally be chosen non-deterministically. Since the expressions embedded in a command sequence may contain free state variables, these must be replaced by the actual state at the point of each command, hence the application of $[\sigma]$ at various occurrences in rules *j* to *p*. Unfortunately, the assignment rule *j* looks more complicated than it is, due to the fact that our heap notation is inherently recursive, something which the assignment command is not.

The three rules which refer to the $\Downarrow$ relation (rules *l* to *n*) use $(\cdots)$ as an abbreviation for the object(s) in focus, to express that evaluation will take place in the context of the complete current heap. Rules *m* and *n* show the difference between asynchronous and synchronous requests, in that the latter rule is only applicable when the recipient is inactive. Furthermore, the coupling between a

(a) $$\frac{\Gamma \triangleright e \Downarrow \Gamma' \triangleright e'}{\Gamma \oplus v = e \triangleright v \Downarrow \Gamma' \oplus v = e' \triangleright e'}$$

(b) $\Gamma \triangleright k\ e_1 \ldots e_n \Downarrow \Gamma \triangleright k\ e_1 \ldots e_n$

(c) $\Gamma \triangleright \backslash v \to e \Downarrow \Gamma \triangleright \backslash v \to e$

(d) $$\frac{\Gamma \triangleright e \Downarrow \Gamma' \triangleright \backslash v \to e_v \quad \Gamma' \oplus v = e' \triangleright e_v \Downarrow \Gamma'' \triangleright e''}{\Gamma \triangleright e\ e' \Downarrow \Gamma'' \triangleright e''}$$

(e) $$\frac{\Gamma \oplus bs \triangleright e \Downarrow \Gamma' \triangleright e'}{\Gamma \triangleright \texttt{let}\ bs\ \texttt{in}\ e \Downarrow \Gamma' \triangleright e'}$$

(f) $$\frac{\Gamma \triangleright e \Downarrow \Gamma' \triangleright k\ e_1 \ldots e_n \quad \Gamma' \oplus v_1 = e_1 \cdots v_n = e_n \triangleright e' \Downarrow \Gamma'' \triangleright e''}{\Gamma \triangleright \texttt{case}\ e\ \texttt{of}\ \cdots k\ v_1 \ldots v_n \to e' \cdots \Downarrow \Gamma'' \triangleright e''}$$

(g) $$\frac{\Gamma \triangleright e \Downarrow \Gamma' \triangleright \{ cs \mid \cdots v_n \cdots \mid \sigma \} \quad \Gamma' \triangleright v_n \Downarrow \Gamma'' \triangleright e'}{\Gamma \triangleright e.s_n \Downarrow \Gamma'' \triangleright e'}$$

(h) $\Gamma \triangleright \texttt{action}\ v\ cs \Downarrow \Gamma \triangleright \texttt{action}\ v\ cs$

(i) $\Gamma \triangleright \{ cs \mid vs \mid \sigma \} \Downarrow \Gamma \triangleright \{ cs \mid vs \mid \sigma \}$

(j) $$\Gamma \oplus v = \{\texttt{set}\ bs : cs \mid vs \mid \sigma\} \Longrightarrow \Gamma \oplus bs' \oplus v = \{ cs \mid vs \mid \sigma \overrightarrow{\oplus} \sigma' \}$$
$$\begin{array}{rrcl} & bs & \equiv & v_1 = e_1 \cdots v_n = e_n \\ \text{where} & bs' & \equiv & v_1[\sigma'] = e_1[\sigma] \cdots v_n[\sigma'] = e_n[\sigma] \\ & \sigma' & \equiv & \textsc{new}(v_1 \ldots v_n) \end{array}$$

(k) $\Gamma \oplus v = \{\texttt{let}\ bs : cs \mid vs \mid \sigma\} \Longrightarrow \Gamma \oplus bs[\sigma] \oplus v = \{ cs \mid vs \mid \sigma \}$

(l) $$\frac{\Gamma \oplus (\cdots) \triangleright e[\sigma] \Downarrow \Gamma' \oplus (\cdots) \triangleright k\ e_1 \ldots e_n}{\Gamma \oplus v = \{\texttt{case}\ e\ \texttt{of}\ \cdots k\ v_1 \ldots v_n \to cs' \cdots : cs \mid vs \mid \sigma\} \Longrightarrow \Gamma' \oplus \begin{array}{l} v = \{ cs' +\!\!+ cs \mid vs \mid \sigma \} \\ v_1 = e_1 \ \cdots \ v_n = e_n \end{array}}$$

(m) $$\frac{\Gamma \oplus (\cdots) \triangleright e[\sigma] \Downarrow \Gamma' \oplus (\cdots) \triangleright \texttt{action}\ v'\ cs''}{\Gamma \oplus \begin{array}{l} v = \{\texttt{req}\ es \oplus e : cs \mid vs \mid \sigma\} \\ v' = \{ cs' \mid vs' \mid \sigma' \} \end{array} \Longrightarrow \Gamma' \oplus \begin{array}{l} v = \{\texttt{req}\ es : cs \mid vs \mid \sigma\} \\ v' = \{ cs' +\!\!+ cs'' \mid vs' \mid \sigma' \} \end{array}}$$

(n) $$\frac{\Gamma \oplus (\cdots) \triangleright e[\sigma] \Downarrow \Gamma' \oplus (\cdots) \triangleright \texttt{action}\ v'\ cs'}{\Gamma \oplus \begin{array}{l} v = \{\texttt{req}\ gs \oplus v'' \leftarrow e : cs \mid vs \mid \sigma\} \\ v' = \{\_ \mid vs' \mid \sigma' \} \end{array} \Longrightarrow \Gamma' \oplus \begin{array}{l} v = \{\texttt{req}\ gs \oplus v'' \leftarrow v' : cs \mid vs \mid \sigma\} \\ v' = \{ cs' \mid vs' \mid \sigma' \} \end{array}}$$

(o) $$\Gamma \oplus \begin{array}{l} v = \{\texttt{req}\ gs \oplus v'' \leftarrow v' : cs \mid vs \mid \sigma\} \\ v' = \{\texttt{reply}\ e : cs' \mid vs' \mid \sigma' \} \end{array} \Longrightarrow \Gamma \oplus \begin{array}{l} v = \{\texttt{req}\ gs : cs \mid vs \mid \sigma\} \\ v' = \{ cs' \mid vs' \mid \sigma' \} \\ v'' = e[\sigma'] \end{array}$$

(p) $\Gamma \oplus v = \{\texttt{req}\ \_ : cs \mid vs \mid \sigma\} \Longrightarrow \Gamma \oplus v = \{ cs \mid vs \mid \sigma \}$

Figure 5: Operational semantics of Omelett

synchronous sender and the receiver must be maintained until the receiver issues a `reply`; to achieve this we apply a small trick which replaces the sender's generating action expression with a reference to the receiver (v'). This reference (which can never evaluate to an action expression) uniquely determines the objects which may participate in a `reply` data transfer (rule o).

The non-deterministic choice which must generally be made in the $\Longrightarrow$ rules is our current definition of the intuitive parallelism between objects. This definition is admittedly of a rather coarse grain, since it does not allow any change of focus during expression evaluation. A definition based on a small-step evaluation semantics would of course be more coherent with a real parallel implementation (albeit at the price of increased complexity), and such a reformulation is a natural target of future work.

We have not yet undertaken a formal proof that the semantics of lazy expression evaluation is left unaffected by its inclusion in an object-oriented context. Still, we think that our claim is informally justified by the following observations:

- Rules a to f are essentially a copy of those defined by Launchbury [Lau93], who proves that his semantics is computationally adequate with respect to a denotational semantics for lazy functional languages.
- The action and object expressions are canonical (rules h and i), which means that they could have been modelled by the constructors of an ordinary datatype. The action expression is even degenerate, since action values cannot be scrutinized within the expression sublanguage.
- The selection expression discards the state-dependent components of object nodes completely (rule g), thus the dot-operator can be considered identical to purely functional record selection.
- All the object manipulating rules (j to p) preserve the attributes component of object nodes, as well as the contents of any node which is not an object.
- Updates due to rule a cannot affect the consistency of rules j to p, since the object nodes in focus are already on canonical form.

5 Related work

Our work is to a large extent inspired by a paper by MacLennan, who investigates different applications of objects and values in programming laguages [Mac82]. His discussion is very general, so the question of what a concrete, combined language should look like is left open. Unfortunately, we are not aware of any further work by MacLennan in this direction. *Reactive* is a term coined by Pnueli to classify programs which are not purely transformational [Pnu86], and we believe our use of the word is a rather faithful extension of his view. Likewise, we borrow the idea that program execution can be seen as a *simulation* from the tradition of object-oriented languages, starting with Simula [DMN70].

Hudak and Sundaresh [HS88], Carlsson and Hallgren [CH93], and Gordon [Gor93], address basically the same problems as we do in their work on functional I/O systems. The solutions they present can all be characterized as (collections of) purely functional programs, managed by an anonymous, unpure operating system which handles updates to the system state and supports non-deterministic merging of events. Our approch can be said to move this "unpure" part inside the programs, but in such a way that the purity of expression evaluation is maintained. We see several advantages by this move: state-manipulation is more direct and intuitive than in a continuation- or stream-based style, there is no need for an additional operational interpretation of the values computed by a program,

and we expect programs to be easier to compose and extend when even the "operating system" is in the hands of the programmer. Of course this comes at the price of introducing new primitive ideas such as objects, actions, and commands, but we believe that an adaption of these concepts will actually prove beneficial in the construction of large, reactive programs.

Current research in extending functional languages with state is mostly motivated by the need for efficient encodings of inherently imperative algorithms. In the context of lazy evaluation, this field is represented by a succession of achievements, headed by the work of Launchbury and Peyton Jones [LPJ94, ORH93, Hud92, SRI91]. These solutions focus on how to encapsulate state-manipulating threads inside apparently pure functions; to the extent that I/O is considered, it is viewed as a special case of an imperative calculation, analogous to the treatment of I/O in traditional languages. Since our goal is of a very different nature, our approach notably lacks the ability to treat state-threads as functions. Still, we may identify several similarities between our reactive model and the work of [LPJ94]: laziness and referential transparency is preserved for all expressions, state references can be part of data-structures manipulated by ordinary functions, and there are no restrictions imposed on the polymorphic type-system. The most obvious advantage of the reactive model is its ability to partition the global system state into self-contained, parallel objects.

There are several languages that provide mixtures of functional, object-oriented, and parallel programming. Holmström presents a language for parallel functional programming with an asymmetric structure similar to ours [Hol83]. Although based on the *almost*-functional language ML, it has the important property that the meaning of expressions is preserved through state changes on the outer, parallel level. Holmström's language is different from ours, though, in that it models state implicitely via parameterized behaviour expressions, and the object-oriented notion of unique objects is replaced by anonymous behaviours communicating over unique *channels*. *CML* and *Erlang* are two examples of parallel-functional-hybrids which have been successfully used in real-life applications [Rep92, AVW93]. However, both these languages mix expression evaluation with communication primitives, so they must resort to strict evaluation semantics. Still, the notion of higher-order concurrency in CML has some interesting similarities to our treatment of actions as values.

From the object-oriented tradition stems *POOL*, with its notion of parallel objects [Ame85]. Functions are not primitive in POOL, however, instead these are defineable as actions without side-effects, at the responsibility of the programmer. This route is also taken in two languages that mix object-orientation with concepts from functional programming: *LOOP*, and *UFO* [ESTZ94, Sar93]. The advantage of these approaches is of course smaller language definitions, but as a consequence, many benefits of functional languages are lost. Whether our approach represents an acceptable compromise between language complexity and desirable program properties remains to be seen.

References

[Ame85] P. America. Definition of the programming language POOL-T. Doc Nr. 0091, Philips Laboratories, Eindhoven, the Netherlands, 1985.

[AVW93] J. Armstrong, R. Virding, M. Williams. *Concurrent Programming in Erlang.* Prentice-Hall International, 1993.

[CH93] M. Carlsson, T. Hallgren. Fudgets — A Graphical User Interfaces in a Lazy Functional Language. In *Proc. Functional Programming and Computer Architecture, Copenhagen*, ACM Press, 1993.

[DMN70] O.J. Dahl, B. Myhrhaug, K. Nygaard. Simula 67 common base language. N.S-22, Norwegian Computer Center, Oslo, 1970.

[ESTZ94] J. Eifrig, S. Smith, V. Trifonov, A. Zwarico. Applications of OOP Type Theory: State, Decidability, Integration. In *SIGPLAN Notices*, vol 29, nr 10, 1994, pp 16-30.

[Gor93] A. Gordon. An Operational Semantics for I/O in a Lazy Functional Language. In *Proc. Functional Programming and Computer Architecture, Copenhagen*, ACM Press 1993.

[HJW+91] P. Hudak, S. Peyton Jones, P. Wadler, et al. Report on the Programming Language Haskell. Yale University, 1991.

[Hol83] S. Holmström. PFL: A Functional Language for Parallel Computing. PMG Technical Report No. 7, Chalmers University of Technology, 1987.

[Hud92] P. Hudak. Mutable abstract datatypes. Research report YALEU/DCS/RR-914, Yale University, December 1992.

[HS88] P. Hudak, R. Sundaresh. On the expressiveness of purely functional I/O systems. Research report YALEU/DCS/RR-665, Yale University, December 1988.

[Hyp87] K. Hyyppä. Optical Navigation System using Passive Identical Beacons. In *Proc. Intelligent Autonomous Systems, Amsterdam*, North-Holland, 1987.

[Lau93] J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. Principles of Programming Languages, Charleston*, ACM Press, January 1993.

[LPJ94] J. Launchbury, S.L. Peyton Jones. Lazy Functional State Threads. In *Proc. Programming Language Design and Implementation, Orlando*, ACM Press, 1994.

[Mac82] B.J. MacLennan. Values and Objects in Programming Languages. In *SIGPLAN Notices*, Vol. 17, No. 12, 1982, pp 70-80.

[Nor94] J. Nordlander. Omelett — a Language for Reactive Programming. Licentiate Thesis, Department of Computing Sciences, Chalmers University of Technology, 1994.

[ORH93] M. Odersky, D. Rabin, P. Hudak. Call-by-name, assignment, and the lambda calculus. In *Proc. Principles of Programming Languages, Charleston*, ACM Press, January 1993.

[PJW93] S.L. Peyton Jones, P. Wadler. Imperative functional programming. In *Proc. Principles of Programming Languages, Charleston*, ACM Press, January 1993.

[Pnu86] A. Pnueli. Applications of Temporal Logic to the specification and verification of reactive systems: a survey of current trends. In *Current Trends in Concurrency*, eds. J.W. de Bakker, et al, LNCS 224, Springer Verlag, 1986.

[Rep92] J. Reppy. Higher-order concurrency. Ph.D. Thesis, Cornell University, 1992.

[Sar93] J. Sargeant. Uniting Functional and Object-Oriented Programming. In *Proc. 1st JSSST International Symposium on Object Technologies for Advanced Software, Kanazawa, Japan*, eds. S. Nishio, A. Yonezawa, LNCS 742, Springer Verlag, 1993.

[SRI91] V. Swarup, U.S. Reddy, E. Ireland. Assignments for Applicative Languages. In *Functional Programming Languages and Computer Architecture, Boston*, ed. Hughes, LNCS 523, Springer Verlag, 1991.

Inferring Effect Types in an Applicative Language with Asynchronous Concurrency

Josva Kleist Martin Hansen
Bo Jensen Hans Hüttel

Department of Mathematics and Computer Science Aalborg University
Fredrik Bajersvej 7E
9220 Aalborg Ø
Denmark
Email: {kleist,mahans,bjens,hans}@iesd.auc.dk

December 21, 1994

Keywords: Type inference, process calculi, effect type, asynchrony.

1 Introduction

In recent years, a number of concurrent programming languages have been based around applicative languages. Best known are CML [Rep92], Facile [GMP89] and LCS [Ber93], all extensions of Standard ML [MTH90]. One would of course want to extend the type inference paradigm of Standard ML [Mil78] to cope with the new concurrency constructs.

Recently, the effect type discipline proposed by Lucassen [Luc87] and Talpin and Jouvelot [TJ92] has attracted a lot of interest as a possible means of capturing these non-applicative aspects, the idea being to associate with an expression not only its type but also the side effects of its evaluation.

Nielson and Nielson have proposed a monomorphic effect type discipline for CML [NN93]. Here the effect of executing a CML expression was a process calculus term. In [Tho93] Thomsen proposed a very different effect polymorphic effect type discipline for the Facile language, and independently, in [BD94] Bolignano and Debabi gave a very similar type discipline for a subset of Concurrent ML – here, the effect of a program is the *sort* of the program, i.e. the set of communication channels used and all causality is absent.

In this paper we consider λinda, a concurrent applicative language based on a rather different concurrency paradigm, namely the Linda concept of Gelernter and Bernstein [GB82], and give a type system where effects are described by expressions in a process calculus.

The type inference system proposed in this paper introduces a novel combination of two previous solutions that have been used to deal with a problem peculiar to applicative programming languages, that of determining the polymorphic effect type of functions. Should one use subtyping or effect polymorphism? We argue that a combination is needed. We give a sound type reconstruction algorithm based around our idea.

Proofs of all results stated in the paper are found in [KJH94].

2 λinda

λinda adds the Linda concept of [GB82] to an applied call-by-value λ-calculus. Linda is based on the notion of a *tuple space*, a shared memory visible to all processes. A tuple space is a multiset of tuples – a tuple consists either of data or evaluating subprocesses. Each process in the program can modify the tuple space by depositing or removing tuples.

As tuples were simply a particular data structure introduced in the original implementation of Linda, we generalize the notion to that of *processes*. In λinda any meaningful expression can be deposited in a *process space* which is a multiset of expressions. Four primitives are used to manipulate the process space: in removes expressions, rd copies expressions from the process space. out deposits values and eval deposits expressions to be evaluated. Communication through a process space is *asynchronous* – processes only block if the expression they want to input does not exist in the process space. The choice of which expression to retrieve is made by *pattern matching*; λinda extends Linda in that patterns are first-class objects. Further, we extend λinda with multiple process spaces to gain the possibility of encapsulation.

2.1 Syntax

The syntax of λinda expressions is

$$\begin{array}{lcl} e & ::= & v \mid x \mid e_1\, e_2 \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \mathsf{rec}\, x(y){\cdot}e \\ v & ::= & c \mid \lambda x{\cdot}e \mid <c\, v_1 \cdots v_n> \end{array}$$

Values are thus either constants, λ-abstractions or *closures*. The latter handle function constants – see below. Variables x range over the set **Var**. $\mathsf{rec}\, x(y){\cdot}e$ defines a recursive function with parameter y and body e possibly containing recursive calls (occurrences of x).

The constants of λinda have the abstract syntax

$$\begin{array}{lcl} c & ::= & () \mid \mathsf{true} \mid \mathsf{false} \mid n \mid p \mid \mathsf{pair} \mid \mathsf{fst} \mid \mathsf{snd} \\ & \mid & \mathsf{nil} \mid \mathsf{cons} \mid \mathsf{head} \mid \mathsf{tail} \mid \mathsf{isnil} \mid ? \mid \ \mathsf{in}\ \mid \mathsf{rd} \mid \mathsf{out} \mid \mathsf{eval} \mid \mathsf{makeps} \end{array}$$

The set of constants includes the basic values: () of unit type, the booleans, integers $n \in \mathbb{Z}$, and process space identifiers $p \in$ **PS-id**. The latter cannot be written as literals but only appear as the result of creating a process space by makeps. The set of constants includes operators for making pairs and lists and the operators for communication: '?' is a pattern matching any value, in , rd, out and eval are the Linda primitives, and makeps is used to create new process spaces.

2.2 The Operational Semantics of λinda

The operational semantics of λinda has a sequential level dealing with the evaluation of a single expression and a concurrent level dealing with the evolution of the process spaces.

Sequential Evaluation

The semantics of the sequential evaluation is based on the notion of evaluation contexts of [Rep92]:

$$E \ ::= \ [\,] \mid E\, e \mid v\, E \mid \mathsf{let}\ x = E\ \mathsf{in}\ e \mid \mathsf{if}\ E\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$$

Any λinda expression is a completed evaluation context $E[e]$ containing the subexpression e to be evaluated; the definition of E ensures call-by-value semantics, c.f. the operational rules in Table 1.

[app1]	$E[(\lambda x{\cdot}e)\, v] \xrightarrow{\varepsilon} E[e\{v/x\}]$
[app2]	$E[v_1\, v_2\, \tau_2] \xrightarrow{\varepsilon} E[v_3] \quad \mid if \mid (v_1, v_2, v_3) \in \Delta$
[if1]	$E[\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2] \xrightarrow{\varepsilon} E[e_1]$
[if2]	$E[\ \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2] \xrightarrow{\varepsilon} E[e_2]$
[let]	$E[\ \mathsf{let}\ x = v\ \mathsf{in}\ e] \xrightarrow{\varepsilon} E[e\{v/x\}]$
[rec]	$E[\mathsf{rec}\, x(y){\cdot}e] \xrightarrow{\varepsilon} E[(\lambda y{\cdot}e)\{(\mathsf{rec}\, x(y){\cdot}e)/x\}]$
[in]	$E[<\mathsf{in}\ v_1> v_2] \xrightarrow{\mathsf{in}(v_1,v_2,v_3)} E[v_3]$
[rd]	$E[<\mathsf{rd}\ v_1> v_2] \xrightarrow{\mathsf{rd}(v_1,v_2,v_3)} E[v_3]$
[out]	$E[<\mathsf{out}\ v_1> v_2] \xrightarrow{\mathsf{out}(v_1,v_2)} E[()]$
[eval]	$E[<\mathsf{eval}\ v_1> v_2] \xrightarrow{\mathsf{eval}(v_1,v_2\,())} E[()]$
[makeps]	$E[\ \mathsf{makeps}\ ()] \xrightarrow{\mathsf{makeps}(v)} E[v]$

Table 1: The semantics at the sequential level

These define a labelled transition system with labels l denoting the communication occurring (if any):

$$l ::= \mathsf{in}(v_1, v_2, v_3) \mid \mathsf{rd}(v_1, v_2, v_3) \mid \mathsf{out}(v_1, v_2) \mid \mathsf{eval}(v_1, v_2\,()) \mid \mathsf{makeps}(v) \mid \varepsilon$$

with ε denoting internal computation.

The rule [app2] states that the application of two values (v_1, v_2) can evaluate to a third value v_3 if $(v_1, v_2, v_3) \in \Delta$. Δ, defined in Table 2, only deals with the first argument of the Linda primitives as application of the second argument has a side effect on the whole system and is handled by the rules for Linda primitives.

$\Delta \supset \{$	(pair,	v,	<pair v>),
	(<pair v_1>,	v_2,	<pair v_1 v_2>),
	(fst,	<pair v_1 v_2>,	v_1),
	(snd,	<pair v_1 v_2>,	v_2),
	(in ,	v,	< in v>),
	(eval,	v,	<eval v>)

Table 2: Clauses from the relation Δ dealing with evaluation of function constants

Concurrent Evaluation

A λinda program is a pool of process spaces. A process space $S \in$ **ProcSpace** is a multiset of expressions tagged with their type. $S[e{:}\tau]$ here denotes that $e{:}\tau$ is an element of S. The pool of process spaces W maps process space identifiers to process spaces, i.e. $W \in$ **PS-id**$\hookrightarrow$**ProcSpace**.

The semantics of the concurrent evaluation is shown in Table 3. The evaluation of the Linda primitives within an expression results in a change to a process space. out and eval both result in the addition of an expression to a process space ([PSP-out] and [PSP-eval]). in removes a value

[PSP-elem] $$\frac{e \xrightarrow{\varepsilon} e'}{W\,[p \mapsto S[e : \tau]] \longrightarrow W\,[p \mapsto S[e' : \tau]]}$$

[PSP-make] $$\frac{e \xrightarrow{\mathsf{makeps}(p')} e' \quad p' \notin |dom|(W)}{W\,[p \mapsto S[e : \tau]] \longrightarrow W\,[p \mapsto S[e' : \tau], p' \mapsto \emptyset]}$$

[PSP-out] $$\frac{e \xrightarrow{\mathsf{out}(p_2,v)} e'}{W\,[p_1 \mapsto S[e : \tau]] \longrightarrow W\,[p_2 \mapsto W'(p_2) \uplus \{\!|v : \tau'|\!\}]} \quad \text{where } W'=W[p_1 \mapsto S[E[e':\tau]]]$$

[PSP-eval] $$\frac{e \xrightarrow{\mathsf{eval}(p_2,e'')} e'}{W\,[p_1 \mapsto S[e : \tau]] \longrightarrow W\,[p_2 \mapsto W'(p_2) \uplus \{\!|e'' : \tau'|\!\}]} \quad \text{where } W'=W[p_1 \mapsto S[e':\tau]]$$

[PSP-in] $$\frac{e \xrightarrow{\mathsf{in}(p_2,v_1,v_3)} e' \quad v_2 : \tau_2 \in W(p_2) \quad v_3 = \mathsf{match}(v_1 : \tau_1, v_2 : \tau_2)}{W\,[p_1 \mapsto S[e : \tau]] \longrightarrow W\,[p_2 \mapsto W'(p_2) - \{\!|v_2 : \tau_2|\!\}]} \quad \text{where } W'=W[p_1 \mapsto S[e':\tau]]$$

[PSP-rd] $$\frac{e \xrightarrow{\mathsf{rd}(p_2,v_1,v_3)} e' \quad v_2 : \tau_2 \in W(p_2) \quad v_3 = \mathsf{match}(v_1 : \tau_1, v_2 : \tau_2)}{W\,[p_1 \mapsto S[e : \tau]] \longrightarrow W\,[p_1 \mapsto S[e' : \tau]]}$$

Table 3: The semantics at the concurrent level

if it matches the pattern ([PSP-in]), and the rd primitive acts as the in primitive except that the value is not removed ([PSP-rd]).

Observe that the rules [PSP-out] and [PSP-in] introduce an ordering of the changes: the process space containing the active expression is changed and only then is the process space being manipulated changed, as the two references may refer to the same process space.

Matching

Values are input or read from the process space using *pattern matching*; in λinda patterns are first-class objects and have the type τ pat. A pattern can be either a '?' or a concrete value.

The types of patterns are all-important; here we shall assume the presence of the needed types (as provided by the type reconstruction algorithm of Section 4.4). The predicate match first matches the types of the values and then the values themselves using the auxiliary function vmatch. If successful the result of match is the value to be the result of the in or the rd.

$$\mathsf{match}(v_1 : \tau\text{"}pat\text{"}, v_2 : \tau) \quad = \quad \mathsf{vmatch}(v_1, v_2)$$

where vmatch is defined as

$$
\begin{array}{lcl}
\mathsf{vmatch}(v, v) & = & v \text{ if } v \text{ is not a } \lambda\text{-abstraction} \\
\mathsf{vmatch}(?, v) & = & v \\
\mathsf{vmatch}(v, ?) & = & v \\
\mathsf{vmatch}(\langle \mathsf{pair}\ v_1\ v_2 \rangle, \langle \mathsf{pair}\ v_3\ v_4 \rangle) & = & \langle \mathsf{pair}\ \mathsf{vmatch}(v_1, v_3)\ \mathsf{vmatch}(v_2, v_4) \rangle \\
\mathsf{vmatch}(\langle \mathsf{cons}\ v_1\ v_2 \rangle, \langle \mathsf{cons}\ v_3\ v_4 \rangle) & = & \langle \mathsf{cons}\ \mathsf{vmatch}(v_1, v_3)\ \mathsf{vmatch}(v_2, v_4) \rangle
\end{array}
$$

Note that λ-abstractions can only be matched by '?'.

It is possible to output values containing '?' into the process space. To input a '?' the pattern used must be a pattern for a pattern, i.e. have the type $(\tau\ \mathsf{pat})\ \mathsf{pat}$.

3 A small example

In this section we will demonstrate how to create parallel programs in λinda by creating a parallel quicksort algorithm. The main function in the quicksort algorithm is the function split which splits a list into two parts: one where the elements are smaller than or equal to the pivot element and one where the elements are larger. The function split looks like this:

```
split=λpivot·λlist·
  if isnil list then
    ([],[])
  else
    let
      element=head list
      result=split pivot (tail list)
    in
      if element¡=pivot then
        (element::(fst result),snd result)
      else
        (fst result,element::(snd result))
```

In a sequential quicksort the sort function is called recursively on the splitted parts of the list, after which the parts are concatenated into a single sorted list. A sequential version of quicksort could look like:

```
quicksort=λlist·
  let
    result=split (head list) list
    smaller=quicksort (fst result)
    larger=quicksort (snd result)
  in
    smaller++larger
```

Note that we use '++' as concatenation of lists and for convenience the pivot element is chosen as the first in the list.

The idea in the parallel version is to spawn a new process for each recursive function call. So each time a list is divided into two parts by split two new processes are started to sort the parts.

When the new processes terminate they become passive elements in the process space containing the sorted sublists. Now the original process needs to collect the results in the right order. So we need to know how to find the right parts of the list which we want to combine into the resulting list. To this end each result of the partial result is tagged by a number indicating which part of the list it is, such that each time a split occurs the two new processes are numbered $2n$ and $2n+1$.

```
sorter=λps·λpart·λlist·
  if isnil list then
    (part,list)
  else
    let
      (smaller,larger)=split (head list) list
    in
      eval ps (λu·sorter ps (2*part) smaller);
      eval ps (λu·sorter ps (2*part+1) larger);
      let
        smaller=snd (in ps (2*part,?))
        larger=snd (in ps (2*part+1,?))
      in
        (part,smaller++larger)
```

Finally, we need the main function starting the quicksort:

```
quicksortpar=λlist·snd (sorter (makeps ()) 1 list)
```

This example shows a commonly used principle in λinda programs, which is to tag the values in the processes space such that it is possible to decide which value is the result of which process. In this example the tag is a unique number but often the tag is a string indicating what the value is. This also provides the ability to distinguish between results from different programs/algorithms evaluating in parallel.

4 An effect type system for λinda

In the rest of our paper we shall consider an effect type system for λinda. The ype of an expression is $\tau\&\kappa$, τ being the ordinary type and κ the *effect* of the expression, denoting the communication capabilities of the expression; inspired by work of Nielson and Nielson [NN93, NN94], effects are terms from a process calculus and can be given an operational semantics, giving us a tool to show that the behavioural effect of an expression is indeed sensible.

The types τ have the following syntax:

$$\tau \ ::= \ \mathcal{B} \mid \alpha \mid \tau_1 \times \tau_2 \mid \tau \ \mathsf{list} \mid \tau_1 \xrightarrow{\beta} \tau_2 \mid \tau \ \mathsf{pat} \mid \mathsf{ps}\ \rho$$

where $\alpha \in \mathbf{TypVar}$ denotes type variables.

The syntax of process behaviour types **Bexp** is:

$$\begin{array}{lcl} \beta & ::= & \beta + \beta \mid \beta;\beta \mid \mu b{\cdot}\beta \mid b \mid a \mid \varepsilon \\ a & ::= & \mathsf{out}(\rho,\tau) \mid \mathsf{eval}(\rho,\tau\&\beta) \mid \mathsf{rd}(\rho,\tau) \mid \mathsf{in}(\rho,\tau) \mid \mathsf{makeps}(\rho) \end{array}$$

Here +' denotes nondeterministic choice, ';' is sequencing, $\mu b{\cdot}\beta$ describes recursive behaviour, where b ranges over behaviour variables **BehVar**, $a \in \mathbf{Act}$ is the possible actions, and ε denotes the empty behaviour. $\rho \in \mathbf{Reg}$ denotes regions.

Regions, though not essential to our approach, facilitate the semantics of behaviour types presented in Section 4.5. A region intuitively corresponds to the notion of process space – a region is a set of "program points" stating which process spaces a Linda primitive have access to. A region identifier should be thought of as being associated with occurrences of makeps. Regions ρ are given by:

$$\rho \ ::= \ i \mid r \mid \rho_1 \cup \rho_2$$

where $i \in \mathbf{RegId}$ denotes the region identifiers, $r \in \mathbf{RegVar}$ denotes region variables and '$\cup$' denotes the union of regions.

4.1 Determining the behaviour of a function

In [NN93] *subtyping* was used to give behaviour types to CML in a monomorphic setting where the programmer has to state the type of the arguments to functions; whereas *polymorphism* was employed in [NN94]. To see which solution is the most appropriate consider the following two problems.

The Argument Problem

The *argument problem* arises from the fact that functions bind behaviour. If the argument of a function is itself a function, the behaviour of the function may depend on the behaviour of the argument. A related problem is that of the latent behaviour of a function input from a process space. At compile time we have no simple way of determining this behaviour.

To see how the argument problem is handled using subtyping consider this expression

$$\lambda \mathsf{f}{\cdot}(\mathsf{f}\ (\mathsf{in}\ \mathsf{ps}(? : \mathsf{int})){+}7$$

The type of this expression should be $(\mathsf{int} \xrightarrow{\beta'} \mathsf{int}) \xrightarrow{\beta} \mathsf{int}$ where the behaviour β of the function should contain the behaviour β' of f. Since we want an upper bound for the possible communication we have to assume that the argument has every possible behaviour. If we let ∞ denote every possible behaviour the functions get the type

$$(\mathsf{int} \xrightarrow{\infty} \mathsf{int}) \xrightarrow{\mathsf{in}\ (\rho,\mathsf{int});\infty} \mathsf{int}$$

In the case of higher-order functions, the application of an argument yields another function. This function also has a bound behaviour but now we can be more specific about it as we know the behaviour of the argument given. Therefore we have to keep track of the origins of the ∞'s in the effect when dealing with successive applications and instantiate them correspondingly. This amounts to using the ∞'s as *polymorphic effect variables*. By using polymorphism at the level of effects, we can easily solve the argument problem- – we use a generic behaviour variable representing the unknown behaviour contained in f. The type of our expression becomes

$$\forall b{\cdot}(\mathsf{int} \xrightarrow{b} \mathsf{int}) \xrightarrow{\mathsf{in}\ (\rho,\mathsf{int});b} \mathsf{int}$$

The Unification Problem

The *unification problem* arises when a function takes more than one argument with the same type but different effects. For instance cons is given the type:

$$\forall\alpha{\cdot}\alpha \xrightarrow{\varepsilon} \alpha\ \mathsf{list} \xrightarrow{\varepsilon} \alpha\ \mathsf{list}$$

If α is instantiated to a function type then we need to handle the bound behaviours. To demand that the functions in a list all have exactly the same effect would make some programs ill-typed that would be well-typed without effect types. Ignoring the effect will not work, for if we extract and apply a function from the list, we need to know its effect.

The unification problem is easily handled with subtyping. Consider the following two expressions:

$$\lambda\mathsf{w}{\cdot}\ \mathsf{out\ ps\ 7} : \alpha \xrightarrow{\mathsf{out}(\rho,\mathsf{int})} \mathsf{unit}$$
$$\lambda\mathsf{w}{\cdot}\mathsf{in\ ps}'(?:\mathsf{unit}) : \alpha \xrightarrow{\mathsf{in}\,(\rho',\mathsf{unit})} \mathsf{unit}$$

If these two functions should both occur in a list we need to give them the same type, i.e. find an upper bound for their behaviour:

$$\alpha \xrightarrow{\beta} \mathsf{unit} \qquad \text{where } \beta = \mathsf{out}(\rho,\mathsf{int}) + \mathsf{in}(\rho',\mathsf{unit})$$

The unification problem is not handled as elegantly using polymorphism. If we were to give the two expressions the same behaviour type using polymorphism we must extend each behaviour with a behaviour variable and unify them. This means that all functions including those with an empty behaviour need a behaviour variable. For example head gets the type

$$\forall\alpha\forall b{\cdot}\alpha\ \mathsf{list} \xrightarrow{\varepsilon+b} \alpha$$

We find this solution clumsy because the type will contain behaviour variables which often will not need to be instantiated. In the above example b might never be instantiated.

We are led to conclude that neither of the existing solutions will suffice in itself; subtyping solves the unification problem elegantly whereas polymorphism solves the argument problem elegantly. For λinda we combine the solutions using polymorphism to describe how the behaviour of a function argument is related to the behaviour of the function applied and using subtyping to handle arguments related by the same type variable.

4.2 Ordering Behaviour Types

Our subtype ordering expresses that one behaviour has fewer communication capabilities than another. As we saw above, we need a supremum operator $+$.

Definition 4.1
We define $(\mathbf{Bexp}, \leq)$ *to be the least preorder satisfying:*

[Supremum]	$\beta_1 \leq \beta_1 + \beta_2$	$\beta_2 \leq \beta_1 + \beta_2$
	$\beta + \beta \leq \beta$	
[Structural]	$\dfrac{\beta_1 \leq \beta_1' \quad \beta_2 \leq \beta_2'}{\beta_1;\beta_2 \leq \beta_1';\beta_2'}$	$\dfrac{\beta_1 \leq \beta_1' \quad \beta_2 \leq \beta_2'}{\beta_1 + \beta_2 \leq \beta_1' + \beta_2'}$
[Identity]	$\beta \leq \beta;\varepsilon$	$\beta;\varepsilon \leq \beta$
	$\beta \leq \varepsilon;\beta$	$\varepsilon;\beta \leq \beta$
[Unfold]	$\mu b{\cdot}\beta \leq \beta\{\mu b{\cdot}\beta/b\}$	$\beta\{\mu b{\cdot}\beta/b\} \leq \mu b{\cdot}\beta$

Moreover we need an ordering on regions:

Definition 4.2
We define $(\mathbf{Reg}, \leq)$ *as the least preorder for which* $\cup$ *acts as a supremum.*

Now we can define an ordering on types taking latent behaviours and regions into account.

Definition 4.3
We define $(\mathbf{Typ}, \leq)$ *as the least preorder satisfying*[1]*:*

$$
\begin{array}{rcl@{\quad}l}
\tau & \leq & \tau \text{ pat} & \\
\tau_1 \text{ list} & \leq & \tau_2 \text{ list} & \text{if } \tau_1 \leq \tau_2 \\
\tau_1 \times \tau_2 & \leq & \tau_3 \times \tau_4 & \text{if } \tau_1 \leq \tau_3 \wedge \tau_2 \leq \tau_4 \\
\tau_1 \xrightarrow{\beta_1} \tau_2 & \leq & \tau_3 \xrightarrow{\beta_2} \tau_4 & \text{if } \tau_3 \leq \tau_1 \wedge \tau_2 \leq \tau_4 \wedge \beta_1 \leq \beta_2 \\
\text{ps } \rho_1 & \leq & \text{ps } \rho_2 & \text{if } \rho_1 \leq \rho_2
\end{array}
$$

4.3 Typing Rules

Judgements are written $\Gamma \vdash e : \tau \& \beta$ and state that in type environment Γ the expression e has type τ and when evaluated the behaviour β. Since we have strict evaluation, variables and constants have no behaviour (except behaviour bound in a function type), so type environments have functionality $\Gamma : \mathbf{Var} \rightarrow \mathbf{TypSch}$. The type inference rules are given in Table 4.

[abs] specifies that abstraction captures the behaviour associated with the body. [app] reflects the call-by-value semantics; the effect of evaluating an application is the effect of evaluating the function to be applied, followed by the effect of evaluating the argument and finally the effect of the application itself. Conditionals are handled by [cond] – if the branches of a conditional have different behaviours, the subsumption rule is used to give them the same behaviour. [rec] is similar to the abstraction rule except that we have to extend the type environment with assumptions about the type of the recursion variable.

We let $t \in \mathbf{TypSch}$ denote type schemes:

$$t ::= \tau \mid \forall \alpha{\cdot}t \mid \forall b{\cdot}t \mid \forall r{\cdot}t$$

We write $\tau \prec t$ to state that the type τ is an instance of the type scheme t. The type schemes of the constants are given by the function TypeOf (Table 5).

[1] For simplicity we have excluded some rules needed to handle wildcards, see [KJH94]

[const] $\Gamma \vdash c : \tau \& \varepsilon$ if $\tau \prec \mathsf{TypeOf}(c)$

[var] $\Gamma \vdash x : \tau \& \varepsilon$ if $\tau \prec \Gamma(x)$

[sub] $$\frac{\Gamma \vdash e : \tau_1 \& \beta_1}{\Gamma \vdash e : \tau_2 \& \beta_2} \quad \text{if } \tau_1 \leq \tau_2 \text{ and } \beta_1 \leq \beta_2$$

[add-hyp] $$\frac{\Gamma \vdash e : \tau \& \beta}{\Gamma[x \mapsto t] \vdash e : \tau \& \beta} \quad \text{if } x \notin {}^*FV^*(e)$$

[abs] $$\frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2 \& \beta}{\Gamma \vdash \lambda x \cdot e : \tau_1 \xrightarrow{\beta} \tau_2 \& \varepsilon}$$

[app] $$\frac{\Gamma \vdash e_1 : \tau_2 \xrightarrow{\beta_3} \tau_1 \& \beta_1 \qquad \Gamma \vdash e_2 : \tau_2 \& \beta_2}{\Gamma \vdash e_1\, e_2 : \tau_1 \& \beta_1; \beta_2; \beta_3}$$

[cond] $$\frac{\Gamma \vdash e : \mathtt{bool} \& \beta' \quad \Gamma \vdash e_1 : \tau \& \beta \quad \Gamma \vdash e_2 : \tau \& \beta}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau \& \beta'; \beta}$$

[let] $$\frac{\Gamma \vdash e_1 : \tau_1 \& \beta_1 \quad \Gamma[x \mapsto \mathsf{Gen}(\Gamma, \tau_1)] \vdash e_2 : \tau_2 \& \beta_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \& \beta_1; \beta_2}$$

[rec] $$\frac{\Gamma[x \mapsto \tau_1 \xrightarrow{\beta} \tau_2, y \mapsto \tau_1] \vdash e : \tau_2 \& \beta}{\Gamma \vdash \mathsf{rec}\, x(y) \cdot e : \tau_1 \xrightarrow{\beta} \tau_2 \& \varepsilon}$$

[clos] $$\frac{\Gamma \vdash c : \tau_1 \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} \tau_n \& \varepsilon \quad \Gamma \vdash v_i : \tau_i \& \varepsilon}{\Gamma \vdash \langle c\, v_1 \cdots v_m \rangle : \tau_{m+1} \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} \tau_n \& \varepsilon} \quad \text{for } 1 \leq i \leq m < n$$

Table 4: Type inference rules.

c	TypeOf
pair	$\forall\alpha\forall\alpha'\cdot\alpha \xrightarrow{\varepsilon} \alpha' \xrightarrow{\varepsilon} \alpha \times \alpha'$
cons	$\forall\alpha\cdot\alpha \xrightarrow{\varepsilon} \alpha\ \mathsf{list} \xrightarrow{\varepsilon} \alpha\ \mathsf{list}$
in	$\forall r\forall\alpha\cdot\mathsf{ps}\ r \xrightarrow{\varepsilon} \alpha\ \mathsf{pat} \xrightarrow{\mathsf{in}(r,\alpha)} \alpha$
rd	$\forall r\forall\alpha\cdot\mathsf{ps}\ r \xrightarrow{\varepsilon} \alpha\ \mathsf{pat} \xrightarrow{\mathsf{rd}(r,\alpha)} \alpha$
out	$\forall r\forall\alpha\cdot\mathsf{ps}\ r \xrightarrow{\varepsilon} \alpha \xrightarrow{\mathsf{out}(r,\alpha)} \mathtt{unit}$
eval	$\forall r\forall\alpha\forall b\cdot\mathsf{ps}\ r \xrightarrow{\varepsilon} (\mathtt{unit} \xrightarrow{b} \alpha) \xrightarrow{\mathsf{eval}(r,\alpha\&b)} \mathtt{unit}$
makeps	$\forall r\cdot\mathtt{unit} \xrightarrow{\mathsf{makeps}(r)} \mathsf{ps}\ r$

Table 5: The types of a sample of the constants. In makeps we will only instantiate r to region identifiers.

The rule [let] makes use of the function Gen which generates a type scheme given a type environment and a type by quantifying over variables not bound in Γ:

$$\begin{aligned}\mathsf{Gen}(\Gamma,\tau) \;=\; & \mathsf{let}\ \{\alpha_1,\ldots,\alpha_n,b_1,\ldots,b_m,r_1,\ldots,r_l\} = \{\kappa \mid \kappa \in FV(\tau)\setminus FV(\Gamma)\} \\ & \mathsf{in}\ \forall\alpha_1\ldots\forall\alpha_n\forall b_1\ldots\forall b_m\forall r_1\ldots\forall r_l\cdot\tau\end{aligned}$$

where $\kappa \in \mathbf{TypVar} \cup \mathbf{BehVar} \cup \mathbf{RegVar}$.

The type system has the property that the type is preserved during evaluation.

Theorem 4.4 (Local Preservation)
If $\vdash e : \tau\&\beta$ *and* $e \xrightarrow{\varepsilon} e'$ *then* $\vdash e' : \tau\&\beta$.

4.4 A type reconstruction algorithm

Our type reconstruction algorithm $\mathcal{T}$ extends the standards ideas of [Mil78] – we get a type-behaviour pair using the behaviour reconstruction algorithm $\mathcal{W}_\beta$:

$$\mathcal{T}(e) = (\tau,\beta) \ \text{ where } \ (\Theta,\tau,\beta) = \mathcal{W}_\beta(\mathsf{nil},e)$$

Here nil is the empty type environment and Θ is a substitution on type and behaviour variables found by an auxiliary unification algorithm $\mathcal{U}$.

As we have seen, the main problem in inferring behaviour types arises from the handling of higher-order functions. Consider e.g. a list of functions cons (rd ps) (cons (ps)nil); all the functions should have the same type τ but not necessarily the same effect. Naïvely we want the list to have the type $(\alpha\ \mathsf{pat} \xrightarrow{\mathsf{rd}(\rho,\alpha)+\mathsf{in}(\rho,\alpha)} \alpha)\ \mathsf{list}$.

To see how this can be obtained consider the first part cons (rd ps). This must be given the type $(\alpha\ \mathsf{pat} \xrightarrow{b} \alpha)\ \mathsf{list} \xrightarrow{\varepsilon} (\alpha\ \mathsf{pat} \xrightarrow{\mathsf{rd}(\rho,\alpha)+b} \alpha)\ \mathsf{list}$.

In a traditional type system cons has the type $\alpha \to \alpha\ \mathsf{list} \to \alpha\ \mathsf{list}$. This is not adequate here, as the α's are not instantiated to the same type. We therefore parameterize the α's with behaviour variables:

$$\mathsf{TypeOf}(\mathsf{cons}) \;=\; \forall\alpha\forall b\forall b'\cdot\alpha(b) \xrightarrow{\varepsilon} \alpha(b')\ \mathsf{list} \xrightarrow{\varepsilon} \alpha(b+b')\ \mathsf{list}$$

This approach however, will not yield the expected type but rather $(\alpha\ \mathsf{pat} \xrightarrow{\mathsf{rd}(\rho,\alpha)+\mathsf{in}(\rho,\alpha)+b} \alpha)\ \mathsf{list}$ as we have to assign some behaviour to nil.

In the presence of parameterized type variables, simple substitutions are not sufficient as a means of instantiation and the unification algorithm $\mathcal{U}$ must be redefined – only in the case of application should behaviour variables be instantiated whereas behaviours should be summed in other cases. The full details of $\mathcal{U}$ are described in [KJH94].

The type reconstruction algorithm is *sound*:

Theorem 4.5 *If* $\Gamma \vdash e$ *is* closed *and* $(\Theta, \tau, \beta) = \mathcal{W}_\beta(\Gamma, e)$ *then* $\Theta\Gamma \vdash e : \tau \& \beta$

4.5 Semantics of Bexp

We can also give an operational semantics for the behaviour expressions. The semantics is given by a labelled transition system with labels:

$$m \in \mathbf{Act} \cup \{\varepsilon\}$$

We add a new element '♡' to the set of behaviours to indicate termination (no λinda expression has this behaviour.)

The semantics again consists of sequential and a concurrent level. The rules at the sequential level are shown in Table 6. Note that as the choice in '+' does not depend on the communication capabilities of a behaviour, '+' is the *internal choice* of [Hen88] rather than the '+' of CCS [Mil89].

$$\begin{array}{lll}
[\text{act}] & a \xrightarrow{a} \varepsilon & \\
[\text{idle}] & \beta \xrightarrow{\varepsilon} \beta & [\text{seq}]_1 \quad \dfrac{\beta_1 \xrightarrow{m} \beta_1'}{\beta_1;\beta_2 \xrightarrow{m} \beta_1';\beta_2} \\
[\text{term}] & \varepsilon \xrightarrow{\varepsilon} \heartsuit & \\
[\text{choice}]_1 & \beta_1 + \beta_2 \xrightarrow{\varepsilon} \beta_1 & [\text{seq}]_2 \quad \dfrac{\beta \xrightarrow{m} \heartsuit}{\beta;\beta' \xrightarrow{m} \beta'} \\
[\text{choice}]_2 & \beta_1 + \beta_2 \xrightarrow{\varepsilon} \beta_2 & [\text{rec}] \quad \mu b{\cdot}\beta \xrightarrow{\varepsilon} \beta\{\mu b{\cdot}\beta/b\}
\end{array}$$

Table 6: Sequential semantics for behaviour types.

For each element in a process space we have a configuration (β, τ, ρ) and the concurrent level is modelled as a multiset of such configurations. In Table 7 the rules for the concurrent part of the semantics are shown. [PS-elem] states that any element in a process space may evolve separately and [PS-uni] states that any part of the multiset may evolve. The rest of the rules state the usual semantics for the Linda primitives. Note that in the rules [PS-in] and [PS-rd] the regions must have common region identifiers in order to reflect a possible in or rd.

The behaviour type of an expression can be seen as an abstract description of its possible evaluation. We define a relation between labels of the labelled transition system for expressions and labels of the transition system for behaviour expressions:

Definition 4.6 *We define the relation* $\asymp$ *by the following rules:*

$$[\text{PS-elem}] \quad \frac{\beta \xrightarrow{\varepsilon} \beta'}{\{\!|(\beta,\tau,\rho)|\!\} \to \{\!|(\beta',\tau,\rho)|\!\}}$$

$$[\text{PS-in}] \quad \frac{\beta \xrightarrow{\mathsf{in}(\rho'',\tau')} \beta'}{\{\!|(\beta,\tau,\rho),(\heartsuit,\tau',\rho')|\!\} \to \{\!|(\beta',\tau,\rho)|\!\}} \quad \rho' \cap \rho'' \neq \emptyset$$

$$[\text{PS-rd} \quad \frac{\beta \xrightarrow{\mathsf{rd}(\rho'',\tau')} \beta'}{\{\!|(\beta,\tau,\rho),(\heartsuit,\tau',\rho')|\!\} \to \{\!|(\beta',\tau,\rho),(\heartsuit,\tau',\rho')|\!\}} \quad \rho' \cap \rho'' \neq \emptyset$$

$$[\text{PS-out}] \quad \frac{\beta \xrightarrow{\mathsf{out}(\rho',\tau')} \beta'}{\{\!|(\beta,\tau,\rho)|\!\} \to \{\!|(\beta',\tau,\rho),(\heartsuit,\tau',\rho')|\!\}}$$

$$[\text{PS-eval}] \quad \frac{\beta \xrightarrow{\mathsf{eval}(\rho',\tau'\&\beta')} \beta''}{\{\!|(\beta,\tau,\rho)|\!\} \to \{\!|(\beta'',\tau,\rho),(\beta',\tau',\rho')|\!\}}$$

$$[\text{PS-uni}] \quad \frac{PS_1 \to PS_1'}{PS_1 \uplus PS_2 \to PS_1' \uplus PS_2}$$

Table 7: Concurrent evaluation of behaviour types.

$\mathsf{in}(v_1,v_2,v_3)$	$\asymp$	$\mathsf{in}(\rho,\tau)$	*if* $\vdash v_1 : \mathsf{ps}\ \rho\&\varepsilon$ *and* $\vdash v_3 : \tau\&\varepsilon$
$\mathsf{rd}(v_1,v_2,v_3)$	$\asymp$	$\mathsf{rd}(\rho,\tau)$	*if* $\vdash v_1 : \mathsf{ps}\ \rho\&\varepsilon$ *and* $\vdash v_3 : \tau\&\varepsilon$
$\mathsf{out}(v_1,v_2)$	$\asymp$	$\mathsf{out}(\rho,\tau)$	*if* $\vdash v_1 : \mathsf{ps}\ \rho\&\varepsilon$ *and* $\vdash v_2 : \tau\&\varepsilon$
$\mathsf{eval}(v,e)$	$\asymp$	$\mathsf{eval}(\rho,\tau\&\beta)$	*if* $\vdash v : \mathsf{ps}\ \rho\&\varepsilon$ *and* $\vdash e : \tau\&\beta$
$\mathsf{makeps}(v)$	$\asymp$	$\mathsf{makeps}(\rho)$	*if* $\vdash v : \mathsf{ps}\ \rho\&\varepsilon$
ε	$\asymp$	ε	

Using this relation the following theorem states that the semantics of behaviours indeed mimics the semantics of expressions: The type of an expression does not change but the effect does.

Theorem 4.7
If $\vdash e : \tau\&\beta$ *and* $e \xrightarrow{a} e'$ *then there exist* β' *and* a' *such that* $\beta \xrightarrow{a'} \beta'$, $a \asymp a'$ *and* $\vdash e' : \tau\&\beta'$.

As an immediate corollary of Theorem 4.7 we have that when λinda expressions evaluate, their associated behaviours decrease.

Corollary 4.8
If $\vdash e : \tau\&\beta$ *and* $e \xrightarrow{l} e'$ *and* $\vdash e' : \tau\&\beta'$ *then there exists* $m \asymp l$ *such that* $m; \beta' \leq \beta$.

An important difference from the behaviour semantics in [NN93] is that we do not use the ordering of behaviours in the semantics. Instead, the ordering can be recovered from our semantics by a simulation ordering in the sense of [Mil89]. We use $\beta \stackrel{m}{\Longrightarrow} \beta'$ to specify that β can evaluate to β' using exactly one m-transition and zero or more ε-transitions, i.e. $\stackrel{m}{\Longrightarrow} = \stackrel{\varepsilon}{\longrightarrow}^* \stackrel{m}{\longrightarrow} \stackrel{\varepsilon}{\longrightarrow}^*$.

Definition 4.9
A binary relation $S \subseteq \mathbf{Bexp} \times \mathbf{Bexp}$ is a simulation if $(\beta_1, \beta_2) \in S$ implies that if

$$\beta_1 \stackrel{m}{\longrightarrow} \beta_1'$$

then there exists a β_2' such that:

$$\beta_2 \stackrel{m}{\Longrightarrow} \beta_2' \qquad \text{and} \qquad (\beta_1', \beta_2') \in S$$

We write $\beta \preceq \beta'$ if (β, β') is contained in some simulation.

Theorem 4.10 (Soundness)
If $\beta \leq \beta'$ then $\beta \preceq \beta'$.

5 Conclusions and further work

In this paper we present a functional language based on the Linda concept and an effect type system for describing concurrent behaviour. Behaviour types are rich in information about the causality of Linda primitives and which expressions are input and output to the process spaces. The semantics of behaviours shows that the behaviour type of an expression indeed describes its concurrent aspects. Further we have shown that the ordering on behaviours is sound with respect to the given simulation. An open problem at the time of writing is whether the type reconstruction algorithm for behaviour types is complete. We conjecture that this indeed the case. Further, it should be examined whether our type inference rules have the principal type property.

Combining subtyping and polymorphism in order to solve the argument and unification problems elegantly gives the programmer the most intuitive types (although it complicates the type inference algorithm for the behaviour types somewhat). The argument and unification problems also occur in other concurrent functional languages such as CML, Facile and LCS, and the ideas from the type inference algorithm for behaviour types presented here could prove fruitful in these languages.

In [KJH94] we give another, sort-based effect type system. Compared to behaviours, sorts are less descriptive and do not capture the causality of the Linda primitives but only state which expressions are input and output. As there is a straightforward and effective translation of behaviour types to sorts which preserves welltypedness, the combination of this with our type reconstruction algorithm for behaviour effect types provides us with a sort inference algorithm.

Several extensions of the type system presented here can be imagined. In particular, one should consider the possibility of *explicit polymorphism* as this would allow us to communicate polymorphic values directly.

References

[Ber93] Bernard Berthomieu. Programming with Behaviors in an ML framework. The Syntax and Semantics of LCS, April 1993.

[BD94] Dominique Bolignano and Mourad Debabi A Semantic Theory for Concurrent ML *Proceedings of Theoretical Aspects of Computer Software (TACS '94)*, pages 766–785, Springer LNCS 789.

[GB82] David Gelernter and Arthur J. Bernstein. Distributed Communication via Global Buffer. *ACM Symposium on Principles of Distributed Computing*, pages 10–18, August 1982.

[GMP89] A. Giacalone, P. Mishra, and S. Prasad. Facile: A Symmetric Integration of Concurrent and Functional Programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.

[Hen88] Matthew Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[KJH94] Josva Kleist, Bo Jensen, and Martin Hansen. Effect Type Systems for Concurrent Functional Languages with Linda Primitives, January 1994. Student Report — Aalborg University.

[Luc87] J.M. Lucassen. Type and Effects: Towards an Integration of Functional and Imperative Programming, 1987. PhD thesis, Laboratory of Computer Science, MIT.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MTH90] Robin Milner, Mads Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[NN93] Flemming Nielson and Hanne Riis Nielson. From CML to Process Algebras. Technical report, Computer Science Department - Aarhus University, March 1993.

[NN94] Hanne Riis Nielson and Flemming Nielson. Higher-Order Concurrent Programs with Finite Communication Topology. To appear in ACM SIGPLAN-SIGACT, 1994.

[Rep92] John H. Reppy. *Higher-Order Concurrency.* PhD thesis, Department of Computer Science, Cornell University, June 1992.

[Rey89] John C. Reynolds. Syntactic Control of Interference — Part 2. *Automata Languages and Programming, 16th International Colloquium (ICALP '89)*, pages 704–722, Springer LNCS 372.

[Tho93] Bent Thomsen. Polymorphic Sort and Types for Concurrent Functional Programs. Technical report, European Computer-Industry Research Centre, June 93. ECRC-93-10.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. In Anne Copeland, editor, *Proceedings - Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 162–173. IEEE, IEEE Computer Society Press, 1992.

Terminated References and Automatic Parallelization for State Transformers

Peter J. Thiemann, Universität Tübingen, Sand 13, D-72076 Tübingen, Germany

Abstract

The monadic approach to integrate first-class references in purely functional programming languages suffers from escaping references and oversequentialization. An extension of the type system by abstract types prevents references from escaping their scope and facilitates transforming mutable objects into immutable ones in constant time. An enhancement of the types of state transformers by effects uncovers implicit parallelism. A program transformation is makes the implicit parallelism explicit so that it can be exploited by a parallel implementation.

Keywords: state monad, first-class references, abstract types, implicit parallelism.

1 Introduction

Pure lazy functional programming languages perform all computations by function application. They do not have an implicit concept of mutable state and hence do not suffer from side effects. However, the lack of assignments and mutable data structures makes it difficult to formulate efficient algorithms for certain problems.

Fortunately, many algorithms use large data structures like arrays or graphs only in a *single-threaded* manner: after an update to the data structure, its old version is never referenced again. Such a data structure can be updated destructively (by assignments) without introducing side effects. Whenever an assignment is executed, the assignment operation has the unique reference to the modified data structure and no other part of the program has access to it. Recent approaches to augment purely functional programming languages with mutable data structures have introduced abstract datatypes which ensure single-threaded handling of mutable data structures. Among them is the monad of state transformers (state monad, for short) [Wad90, Wad92] which encapsulates the propagation of a state through a whole computation. It provides proper sequencing of I/O operations [PW93] or lazy computations with mutable variables and arrays [Lau93]. In this approach, references to mutable variables and arrays are first-class values which are created and manipulated by primitive operations of the monad.

If the state monad is used to provide proper sequencing of assignment operations, only one incarnation of the state can be active in a given program without the risk of introducing side effects. To see this, suppose a reference created in one incarnation of the state monad is allowed to be returned as a result (and thus to "escape"). If that is possible, an expression e can be constructed which passes the reference to two different incarnations of the state monad such that the value of e depends on the evaluation order. Thus side effects can be introduced unless suitable provisions are taken.

For performing I/O it is perfectly acceptable to have a single global incarnation of the state monad. For performing computations involving references, having just a single incarnation sequentializes the entire program. It is therefore preferable to have local incarnations of state transformers which work independently on local states.

Our Contribution The present work is concerned with the use of state transformers to provide proper sequencing for assignment operations.

1. We propose to use a restricted form of existential types (*downward abstract types*) for the propagated state of state transformers. Each incarnation of a state transformer is provided with its own unique identity through its abstract type. References are first-class objects confined to the local scope of a single incarnation of a state transformer. Programs where a local reference may be returned as a result are rejected by the type checker.

2. State transformers can construct immutable objects: after a mutable object is allocated and changed by assignment operations its final version can simply be declared immutable and re-

leased. This is only safe if there is exactly one live reference to the object. *Upward abstract types*, the dual to downward abstract types, can ensure the required uniqueness property. They are used to type *terminators* which delimit the life-time of a mutable object and may thus transform it into an immutable one without copying it.

3. State transformers impose an arbitrary order on their primitive actions, namely their order in the program text. When multiple mutable objects are manipulated simultaneously this order is often overly conservative. We offer a program transformation that uncovers the inherent parallelism by grouping the primitive operations according to the mutable objects that they act on. The transformation relies on an effect annotation of the type constructor for state transformers. The effect annotation describes the use of mutable objects in the local state.

The mechanism of upward and downward abstract types is quite similar to existentially quantified types. There is a type reconstruction algorithm [Thi93] which handles them like nullary type constructors (cf. the Skolem type constructors used by Läufer and Odersky [LO92] to model abstract types by existentially quantified types as pioneered by Mitchell and Plotkin [MP88]). However, there is no special syntax to introduce a value with an abstract type and type declarations are required to introduce abstract type variables.

Notice that the effect annotation and thus the program transformation to expose parallelism is also usable with the solution to the encapsulation problem of Launchbury and Peyton Jones [LP94]. The transformation also sheds some light on how to program communicating processes with local state in a lazy parallel functional language.

Outline of the Paper Section 2 briefly introduces state transformers and their application to sequence assignments. An example which illustrates the danger of escaping first-class references is given in Section 3. An informal outline of our solution follows. Section 4 introduces the concept of terminators which can be used to transform mutable objects into immutable ones, describes the necessary modifications to the type system, and demonstrates their use with an example. Section 5 analyzes the effects of state transformers and defines a transformation to separate independent parts of state transformers. Sections 6 and 7 define different ways to run the separate parts independently and possibly in parallel. Finally, in Section 8, we discuss what to do when a clean separation in independent parts is not possible. Section 9 discusses related work and Section 10 concludes.

2 State Transformers

A state transformer [Wad90, Wad92] is a function which maps an initial state to a result paired with a final state. Its type definition is:

```
type ST state value = state -> (value, state)
```

This declaration introduces the type constructor ST with two parameters `state` and `value` with the obvious meanings. Here and throughout the whole paper, we make liberal use of Haskell [Has92] syntax in examples. The basic operations on `ST s v` are defined as follows.

```
result :: v -> ST s v
result x = λs -> (x, s)

(▷) :: ST s v -> (v -> ST s w) -> ST s w
m ▷ g = λs -> let (x, s') = m s in g x s'
```

`result x` specifies a state transformer which does not change the state and returns `x` immediately. The *bind* operation ▷ accepts a state transformer `m` and a function `g` from results of `m` to state transformers. `m ▷ g` is a state transformer which first runs `m` with result `x` and then runs the state transformer `g x` obtained by applying `g` to `x`. `result` is a left and right unit with respect to ▷ and ▷ is associative, hence `ST s v` is a monad, for any `s`. We call an executing incarnation of a state transformer a *(state) thread* (cf. [LP94]). A value of type `ST s v` is a *(state) action*. From the definitions we see that the state component is propagated through the whole computation without duplication, therefore the state is single-threaded.

Since `result` and ▷ handle the state in a single-threaded manner it is safe to enhance ST with assignment operations to the state. A state action can be executed with the function `exec` which accepts an initial state value and an action to produce the result after running the action on the initial state (`fst` returns the first component of a pair).

```
exec :: s -> ST s v -> v
exec init action = fst (action init)
```

Peyton Jones, Wadler, and Launchbury [PW93, Lau93] have suggested a more powerful way to use the sequentialization imposed by state transformers. They enhance state transformers by primitive actions to perform an I/O operation, create a reference, read a reference, or assign to a reference. The propagated state is only used to fix the order of their execution.

To execute such a computation it suffices to apply it to an arbitrary (non-bottom) initial state, for example () of type ().

```
run :: ST () v -> v
run action = exec () action
```

With this knowledge a signature for primitive operations on references can be defined [Lau93]. In the following, datatypes whose name ends with a ! denote references to mutable objects. In the signature below, Ref! is the type of references to mutable variables.

```
type Ref! v      -- abstract type of references
type Seq v = ST () v

crtRef ::            v -> Seq (Ref! v)
getRef :: Ref! v ->       Seq v
updRef :: Ref! v -> v -> Seq ()
```

crtRef creates a new variable, getRef reads its contents, and updRef assigns to it. To ensure proper sequencing the functions getRef r and updRef r v must be strict in the propagated state[1], for some r :: Ref! b and v :: b, and also in their reference argument r, but not in the assigned value v. A more precise operational semantics in the form of graph rewriting rules is given in Appendix A. The execution of updRef r v actually assigns the value of v to the variable referenced by r. Two examples are given for computations with references: an action incrRef which accepts a reference to a number and updates it with the incremented value and an action swap which exchanges the values of two references.

```
incrRef :: Ref! Int -> Seq ()
incrRef r = getRef r ▷ λv -> updRef r (v+1)

swap :: Ref! v -> Ref! v -> Seq ()
swap r s =
 = getRef r   ▷ λv -> getRef s   ▷ λw ->
   updRef r w ▷ λ_ -> updRef s v
```

3 Encapsulating References

Since updRef is a real assignment operation, side effects can be introduced through run as follows.

Example 1

```
let r = run (crtRef 0)
    a = incrRef r ▷ λ() -> getRef r
in  (run a, run a)
```

[1] The strictness can be made explicit at the source level by using an algebraic datatype data D x = D x as the state type as in [PW93, LP94]. Since that complicates the types we just state the required strictness.

By referential transparency, the value of this expression should be a pair of equal numbers. However, depending on the context its value is either $(1,2)$ or $(2,1)$. This side effect is due to the reference r :: Ref! Int which escapes from its creating thread run (crtRef 0) and is used in other threads.

In order to avoid escaping references we change the type of run so that the type of the propagated state is abstract. The type is specified with a *downward abstract type* (to be defined below) which is abstracted with the quantifier $\downarrow$. The net effect is that every thread has is own unique state type.

```
run :: ↓s . ST s v -> v
```

As the all-quantification, the quantifier $\downarrow$ is also only allowed at the top-level. The type system still distinguishes between types (without quantification) and type schemes. As in the ML type system the rules for abstraction and application only apply to types.

Furthermore, the type of the manipulated references must be linked to the state of the creating thread. This is done by adding a type variable s to the type of references. The signature is revised as follows.

```
type Ref! s v
-- abstract type of references
-- to variables of type v
-- created by a thread with state type s

crtRef ::               v -> ST s (Ref! s v)
getRef :: Ref! s v ->       ST s v
updRef :: Ref! s v -> v -> ST s ()
```

Downward abstract type variables (quantified with $\downarrow$) are a restricted form of existentially quantified type variables and share their introduction and elimination rules:

$$(\downarrow\mathrm{I})\frac{A \vdash e : \tau[\alpha \mapsto \tau']}{A \vdash e : \downarrow\alpha.\tau} \qquad (\downarrow\mathrm{E})\frac{A \vdash e : \downarrow\alpha.\tau}{A \vdash e : \tau[\alpha \mapsto \underline{\gamma}]}$$

In the elimination rule, $\underline{\gamma}$ is a fresh nullary type constructor which does not occur elsewhere.

If we use the above type ↓s . ST s v -> v for run, Example 1 is not well-typed anymore: the reference r has type Ref! $\underline{\gamma}$ Int, hence a has type ST $\underline{\gamma}$ Int. Moreover, the occurrences of run in the last line have types ST $\underline{\gamma'}$ Int and ST $\underline{\gamma''}$ Int, for $\gamma' \neq \gamma$ and $\gamma'' \neq \gamma$, therefore the applications run a are not well-typed. But existentially quantified type variables are not enough: a function of type $\forall s, v, \ldots.(\mathrm{ST}\ s\ v \to v) \to \ldots$ can take run as a parameter and reconstruct the situation in Example 1 in its body (see [Thi93]). It follows that passing run

as a parameter must be ruled out[2]. The following rule ensures this.

> **Rule 1** In the derivation of a typing judgement $A \vdash e: \tau$ no generic type variable α may be instantiated with a type containing a downward abstract type which is introduced "to the right" of α.

In this context, "left" arbitrarily denotes the function judgement of the application rule, and "right" denotes its argument judgement. Passing `run` as a parameter is impossible: suppose for a contradiction that

$$\dfrac{A \vdash f: \tau' \to \tau'' \qquad \dfrac{\dfrac{A \vdash \texttt{run}: \forall\nu.\downarrow s.\mathrm{ST}\ s\ \nu \to \nu}{A \vdash \texttt{run}: \downarrow s.\mathrm{ST}\ s\ \tau \to \tau}}{A \vdash \texttt{run}: \mathrm{ST}\ \underline{a}\ \tau \to \tau}}{A \vdash f\ \texttt{run}: \tau''}$$

is a valid derivation. The downward abstract type $\underline{a}$ contained in the type instance of `run` must also occur in τ'. But $\underline{a}$ cannot occur in the assumption A, as any application of the introduction rule creates a unique downward abstract type. Hence the derivation for $A \vdash f: \tau' \to \tau''$ must contain some instantiation of a generic type with $\underline{a}$. Since Rule 1 is violated by this instantiation the above derivation cannot be valid.

However, abstract types can travel from left to right in a derivation tree, as illustrated by the following example.

Example 2 The expression `run (result 1)` (which is a prototype of a practically useful expression) is legal since the derivation in Fig. 1 is valid. We only have to check our rule at the point where `result` is introduced. But $\underline{a}$ has been introduced to the left of `result` in the proof tree so the rule is not violated.

The somewhat intuitive Rule 1 can be checked in an implementation in a straightforward way. The type checking algorithm creates new type variables (and new abstract types) by threading a name supply through all its recursive invocations. Typically, the algorithm visits the function part of an application before its argument parts. The names assigned in this manner do reflect an inorder traversal of the derivation tree. Therefore, the check if some α was introduced to the left of the current node of the derivation tree boils down to compare the name of α with the current name supply. The formalization of this is rather technical and details on it are found in a technical report [Thi93].

[2]This effect is achieved in [LP94] by giving `run` the rank-2 type $\forall\alpha.(\forall\beta.\mathrm{ST}\ \beta\ \alpha) \to \alpha$.

4 Introducing Terminators

Arrays in pure functional languages are usually monolithic (the contents cannot be defined incrementally) and immutable (the contents cannot be modified). In Haskell, they are constructed with a function `array` which accepts a size and a list of index-value pairs and returns an immutable array which associates the indices to their respective values in the list. This former primitive function can now safely be written using a state thread: allocate a mutable array, fill it using assignment operations, and let it escape from its creating thread. But the typing of `run` prevents exactly that. Launchbury and Peyton Jones [LP94] have therefore introduced the operation

```
freezeA :: Array! state v -> ST state (Array v)
```

(ignoring the index which is another parameter of Haskell's `Array` datatype) such that `freezeA a` is strict in its `state` argument and in `a` and provides the proper retyping. The implementation of `freezeA` cannot simply be a strict version of `result` which declares its argument immutable: a full copy of the argument array must be performed. Otherwise side effects may be introduced if the array is subsequently assigned to.

If the argument array of `freezeA` is the unique last reference to the array it is safe to omit the expensive copy operation. A further change in the signature and the type system can be used to obtain the necessary uniqueness information. The idea is to introduce *terminators* for references. A terminator is an action that takes a reference as an argument. Its typing ensures that the reference cannot be used after its terminator action has been executed. The type of a terminator contains an *upward abstract type* which behaves dually to downward abstract types:

> **Rule 2** In the derivation of a typing judgement $A \vdash e: \tau$ no generic type variable α may be instantiated with a type containing an upward abstract type which is introduced "to the left" of α.

The introduction and elimination rules for upwards abstract types are the same as for downward abstract types.

To make use of terminators the abstract type `Array!` of references to mutable arrays is augmented with a third parameter. The parameters are `term` for its *terminator type*, `state` to link it with its creating state thread, and `value` for the type of the array elements. The signature has to be changed according to Fig. 2. The intended semantics of the operations is specified as follows. Let `ar :: Array! t s v, i ::`

$$\dfrac{\dfrac{A \vdash \texttt{run}: \forall v.\downarrow s.\texttt{ST}\ s\ v \to v}{A \vdash \texttt{run}: \downarrow s.\texttt{ST}\ s\ \texttt{Int} \to \texttt{Int}}}{A \vdash \texttt{run}: \texttt{ST}\ \underline{a}\ \texttt{Int} \to \texttt{Int}} \qquad \dfrac{A \vdash \texttt{result}: \forall s, v.v \to \texttt{ST}\ s\ v}{A \vdash \texttt{result}: \texttt{Int} \to \texttt{ST}\ \underline{a}\ \texttt{Int}} \quad A \vdash 1: \texttt{Int}$$

$$A \vdash \texttt{result}\ 1: \texttt{ST}\ \underline{a}\ \texttt{Int}$$

$$A \vdash \texttt{run}\ (\texttt{result}\ 1): \texttt{Int}$$

Figure 1: A legal derivation

```
type Array! term state value      -- abstract

createA ::                  Int -> v
                         -> ST s (Array! t s v)
selectA :: Array! t s v -> Int
                         -> ST s v
updateA :: Array! t s v -> Int -> v
                         -> ST s ()
freezeA :: ↑t. Array! t s v -> ST s (Array v)
run     :: ↓s. ST s v -> v
```

Figure 2: Signature for mutable array operations

`Int` and `x :: v`. The action `createA i x` allocates a mutable array with range `[0 .. i-1]` with all entries initialized to `x`. The `i`th entry of array `ar` is obtained with `selectA ar i`. The `i`th entry of `ar` is updated to `x` with `updateA ar i x`. Both `selectA` and `updateA` must be strict in the propagated state `s`, the array reference `ar`, and the index `i`. The action `freezeA` with the above type serves as a terminator for mutable arrays. `freezeA ar` captures the mutable array `ar` and returns an immutable array with the same contents. The typing ensures that `ar` is the unique last reference to the mutable array. Its implementation runs in constant time since the argument array need not be copied.

As an example we provide an implementation of a function `invert` which accepts a permutation π in the form of `perm :: Array Int` (where `perm!!i` $= \pi(i)$, `!!` is Haskell's array indexing operation) and returns the inverse permutation π^{-1} in a newly allocated array.

```
invert :: Array Int -> Array Int
invert perm
 = let n = range a
       help i a
        = if i<n then
             updateA a (perm!!i) i ▷ λ_ ->
             help (i+1) a
          else
             result ()
       bot = bot
   in run (createA n bot ▷ λa ->
           help 0 a ▷ λ_ ->
           freezeA a)
```

In a similar way, it is possible to introduce mutable lists and heaps and perform arbitrary computations on them. Once the desired structure has been constructed the object can be "frozen" as a whole into an immutable object. Notice, that the structure can be circular which makes a copy operation (thus a safe `freeze` without a terminator type) prohibitively expensive.

5 Separating Effects

The primary reason for introducing state transformers is to keep read and write operations to the state in the correct order. However, the actual sequence in which actions are executed depends on an artificial data dependency constructed from the program text. It surely reflects the desired execution order but it is conservative as it can only model a linear order on assignment operations whereas the actual data dependency need only be a partial order. This happens, when a single thread performs computations involving references to multiple mutable data structures, for example, a mutable array and a couple of mutable variables.

The transformation presented below extracts linear dependency chains from a state action and also provides for a way to run them independently, possibly in parallel. The extracted slice encompasses exactly those actions which operate exclusively on a fixed set of references. When there are interdependencies the necessary synchronization can be provided, too.

5.1 Effects for State Transformers

But how can the compiler determine which action belongs into which slice? The type `ST s v` of an action a only describes the possible value of a. It does not describe which variables are read or written by a. We need a description of the *effects* of a. Every result due to the evaluation of an expression which is not reflected in its value is called an effect. Therefore, an *effect system* [LG88] relates an expression not only to

its type but also to its effects. Function abstractions delay the effects of their body until they are applied to an argument. Hence, function arrows are annotated with their latent effects, which may happen during the application of that function. The original motivation to introduce effect systems was the implementation of impure strict functional programming languages on parallel machines. Effect analysis is used for allocating computations to processors and for deciding in which local memory a variable should be allocated.

In the present case, effects can only occur when a state transformer is executed. The only function arrows which must have an effect annotation are hidden in the state transformer type `ST s v = s -> (v, s)`. Thus, we annotate the type constructor `ST` with an *effect description* M. The effect description M for an action a is the set of terminator types of the references that a operates on. If the type `t` is in M then a reads, writes, or creates a reference of type `Ref! t s v`, for some `s` and `v`. The annotations for the primitive actions are as follows:

```
crtRef ::                  v -> ST^{t} s (Ref! t s v)
getRef :: Ref! t s v ->       ST^{t} s v
updRef :: Ref! t s v -> v ->  ST^{t} s v

result :: v -> ST^{} s v
(▷) :: ST^M s v -> (v -> ST^N s w) -> ST^{M∪N} s w
```

`crtRef`, `getRef`, and `updRef` all have an effect on the variable terminated by `t`. The primitive action `result` does not have any effect. The combination `a ▷ λx -> b` of an action `a` with effects M and an action `b` with effects N is an action with effects $M \cup N$. Furthermore there is a subtyping rule which states that `a :: ST^M s v` implies `a :: ST^{M'} s v` for any $M' \supseteq M$. These definitions suffice to determine the least effect of any given action. For a of type `ST^N s v` we set *least-effect*$(a) = M$ where M is the smallest set of types such that `a` has type `ST^M s v`.

5.2 Slicing Actions

Let T be the set of terminator types of a given set of references. Once the effect annotation of an action is known, it can be divided in two subactions or *slices* according to T. One slice comprises all those primitive actions which operate of references in T, while the remaining actions are collected in the second slice. The effect-guided program transformation to extract the slices is presented below.

There are some critical issues in the design of the transformation.

- The separation of the slices drags variables out of their binding scope. The values of these formerly bound variables must be made available.

  This is achieved by having each slice return the values of all bound variables in a tuple. These tuples are then fed back into the other slice with a suitable fixpoint operator.

- The result of the original action is now (part of) the result of one of the slices. It must be recorded which slice returns the original result.

- The laziness properties of the original program should be maintained. Non-trivial expressions should not be duplicated. However, they cannot be evaluated in one of the slices since that introduces unwanted dependencies.

  Sharing of non-trivial expressions is achieved by binding their values to variables with newly introduced `let`-expressions. These `let`s are then lifted out of the slices, so that their scope includes both.

- The transformation need not treat abstractions and simple constants, they are ruled out as their type can never be `ST s v`. Every other expression must be dealt with. However, in this exposition we leave function applications and recursion alone, while compound actions (with a top level ▷), conditionals, and `let`-expressions are split.

- There might be actions with least effect M such that neither $M \subseteq T$ nor $M \cap T = \emptyset$ holds.

  For the moment we assume that all actions can be decomposed (by our transformation) into actions a where either $M \subseteq T$ or $M \cap T = \emptyset$ (M is the least effect of a). Later on, in Section 8, we will drop that assumption.

The transformation makes use of an auxiliary function *split* of type

$$\mathcal{P}(\texttt{Type}) \times \Lambda \times \Lambda \times \Lambda \to \{1,2\} \times \Lambda \times \Lambda \times \Lambda \times \texttt{Binder} \times \Lambda$$

where `Type` is the set of type expressions, Λ is the set of expressions (see Appendix A), and `Binder` is the set of binders, *i.e.* lists of pairs `x = e` with `x` a program variable and $e \in \Lambda$. Binders B occur in the definition part of a `let`-expression: `let` B `in` e.

In the specification of the transformations, ⌈ quotes ⌉ and ⟨ unquotes ⟩ are used to distinguish target program syntax from the meta level. The application *split*$(T, \lceil() \rceil, \lceil() \rceil, a)$ yields a tuple $(i, W_1, W_2, B, r_1, r_2)$ where

- T is the set of *types-of-interest*, the terminator types of the references the actions on which are to be extracted,
- a is the original action expression of type `ST s v`, for some type `v`,
- $i \in \{1, 2\}$ such that r_i is the action delivering the final result of a,
- W_1 and W_2 are nested tuples which contain the bound variables of r_1 and r_2,
- B is a list of binders of the form $X = \Lambda$, which are used to capture shared computations,
- r_1 is the extracted slice of a containing all actions which are annotated by a subset of T, and
- r_2 is the slice with all remaining actions.

The function *split* maintains the following property for a of type $\mathtt{ST}^{\mathsf{M}}\ s\ v$: if $i = 1$ then

$$r_1 :: \mathtt{ST}^{\mathsf{M}_1}\ s\ (v, \mathit{type}(W_1)) \qquad r_2 :: \mathtt{ST}^{\mathsf{M}_2}\ s\ \mathit{type}(W_2)$$

otherwise (if $i = 2$)

$$r_1 :: \mathtt{ST}^{\mathsf{M}_1}\ s\ \mathit{type}(W_1) \qquad r_2 :: \mathtt{ST}^{\mathsf{M}_2}\ s\ (v, \mathit{type}(W_2))$$

where $\mathsf{M}_1 \subseteq \mathsf{T}$ and $\mathsf{M}_2 \cap \mathsf{T} = \emptyset$. Each slice returns the values of its bound variables as a nested tuple and one of them additionally returns the result of the original action.

In a preprocess to *split*, the action a is transformed such that all *bind* operations occur in the form $a \rhd \lambda x.a'$, all bound variables have unique names, and in all expressions of the form `if b then` a_1 `else` a_2 the condition `b` is just a variable. This is achieved by η-expansion, α-conversion, and by introducing auxiliary `let`s.

The rules are given in order of priority as in a real functional program.

The first rule (rule (1) in Fig. 3) for *split* is concerned with normalizing *binds*. Repeated application of the rule isolates a single atomic action on the left-hand side of a top-level $\rhd$. The rule is obviously correct since $\rhd$ is associative.

The following rules (2) and (3) should not be used if the argument action has least effect M where $\mathsf{M} \subseteq \mathsf{T}$ or $\mathsf{M} \cap \mathsf{T} = \emptyset$. In these cases nothing is gained by splitting the action. Rule (7) applies instead.

Rule (2) handles the case where the top-level action is enclosed in a `let`-binding `let x = e in` a. In principle the computation of `e` could be assigned to one of the slices. But it is clearly undesirable to do so because unwanted dependencies between the two slices might be introduced. Therefore the binding `x = e` is appended to the list B of binders and lifted to the top-level. Now `e` can be computed on demand by either of the slices and its value is shared by both.

If the top-level expression is a conditional expression, for example `if b then` a_1 `else` a_2, a conditional expression must be generated in both slices. Since `b` is a variable (by virtue of the preprocessing step) it can be duplicated without duplicating computation. The complicated code in the rule stems from the fact that both branches of a conditional must have the same type. Hence the variable bindings of each branch must also be returned by the other branch. The bound variables of the branch which is not taken are undefined, but never referenced, so that will not cause problems. See rule (3) in Fig. 3.

If the left-hand operand of the top-level $\rhd$ is a `let`-expression, the binding is lifted to the top-level and then handled by rule (2). Pushing the `let` outward does not capture any free variables since distinct variable names have been assumed. The corresponding rule is rule (4) in Fig. 3.

If the left-hand operand of the top-level $\rhd$ splits but is not of the form $a \rhd \lambda x.a'$, it must be split recursively and then prepended to the outcome of the transformed right hand operand. This case applies, for example, when the left-hand operand is a conditional expression. The corresponding rule (5) is found in Fig. 3.

Rule (6) in Fig. 3 treats the case where the left-hand operand of the top-level $\rhd$ cannot be decomposed any further. It must be examined for annotation with a subset of the types-of-interest T and assigned to one of the slices. The bound variable, which might be referenced in the other slice, is added to the respective V. The remaining case, where $\mathsf{M} \not\subseteq \mathsf{T}$ and $\mathsf{T} \cap \mathsf{M} \neq \emptyset$, is covered in Sec. 8.

The final action a (without top-level $\rhd$) must be treated slightly different, since it must be registered which slice delivers the final result. Furthermore B is initialized to the empty list of binders ε. The corresponding rule (7) is shown in Fig. 3.

6 Extracting Terminated Computations

So far, if we are given a set T of types-of-interest (terminator types) and an action a we can slice a in two independent actions a_1 and a_2. It remains to establish the means to run them independently and to feed back the values of the bound variables of a_1 to a_2 and vice versa.

The first goal is met with the operation `interleave :: ST s v -> ST s v` [PW93, LP94].

$$
\begin{array}{l}
\mathit{split}(\mathsf{T}, V_1, V_2, \lceil(\langle a_1\rangle \rhd \lambda\langle x\rangle \to \langle a_2\rangle) \rhd \langle a_3\rangle\rceil) \\
= \mathit{split}(\mathsf{T}, V_1, V_2, \lceil\langle a_1\rangle \rhd \lambda\langle x\rangle \to (\langle a_2\rangle \rhd \langle a_3\rangle)\rceil)
\end{array}
\tag{1}
$$

$$
\begin{array}{l}
\mathit{split}(\mathsf{T}, V_1, V_2, \lceil\texttt{let}\ \langle x\rangle = \langle e\rangle\ \texttt{in}\ \langle a\rangle\rceil) \\
=\ \ \mathit{let}\ (i, W_1, W_2, B, r_1, r_2) = \mathit{split}(\mathsf{T}, V_1, V_2, \langle a\rangle) \\
\quad\ \mathit{in}\ (i, W_1, W_2, \lceil\langle B\rangle; \langle x\rangle = \langle e\rangle\rceil, r_1, r_2)
\end{array}
\tag{2}
$$

$$
\begin{array}{l}
\mathit{split}(\mathsf{T}, V_1, V_2, \lceil\texttt{if}\ \langle b\rangle\ \texttt{then}\ \langle a_1\rangle\ \texttt{else}\ \langle a_2\rangle\rceil) \\
=\ \ \mathit{let}\ (i, W_1, W_2, B, r_1, r_2) = \mathit{split}(\mathsf{T}, \lceil()\rceil, \lceil()\rceil, \langle a_1\rangle) \\
\qquad (i', W_1', W_2', B', r_1', r_2') = \mathit{split}(\mathsf{T}, \lceil()\rceil, \lceil()\rceil, \langle a_2\rangle) \\
\quad\ \mathit{in\ if}\ i = 1\ \mathit{and}\ i' = 1\ \mathit{then} \\
\qquad \mathit{let}\ z\ \mathit{be\ a\ fresh\ variable\ in} \\
\qquad (i, \lceil(z, (\langle W_1\rangle, \langle W_1'\rangle, \langle V_1\rangle))\rceil, \lceil(\langle W_2\rangle, \langle W_2'\rangle, \langle V_2\rangle)\rceil), \\
\qquad \lceil\texttt{if}\ \langle b\rangle\ \texttt{then}\ \langle r_1\rangle \rhd \lambda(z, \langle W_1\rangle) \to \\
\qquad\qquad\qquad \texttt{result}\ (z, (\langle W_1\rangle, \langle W_1'\rangle, \langle V_1\rangle)) \\
\qquad\qquad\ \texttt{else}\ \langle r_1'\rangle \rhd \lambda(\langle W_1'\rangle) \to \\
\qquad\qquad\qquad \texttt{result}\ (z, (\langle W_1\rangle, \langle W_1'\rangle, \langle V_1\rangle))\rceil, \\
\qquad \lceil\texttt{if}\ \langle b\rangle\ \texttt{then}\ \langle r_2\rangle \rhd \lambda(\langle W_2\rangle) \to \\
\qquad\qquad\qquad \texttt{result}\ (\langle W_2\rangle, \langle W_2'\rangle, \langle V_2\rangle) \\
\qquad\qquad\ \texttt{else}\ \langle r_2'\rangle \rhd \lambda(\langle W_2'\rangle) \to \\
\qquad\qquad\qquad \texttt{result}\ (\langle W_2\rangle, \langle W_2'\rangle, \langle V_2\rangle)\rceil) \\
\quad\ \mathit{else}\ \ldots
\end{array}
\tag{3}
$$

$$
\begin{array}{l}
\mathit{split}(\mathsf{T}, V_1, V_2, \lceil(\texttt{let}\ \langle x_1\rangle = \langle e\rangle\ \texttt{in}\ \langle a_1\rangle) \rhd \lambda\langle x_2\rangle \to \langle a_2\rangle\rceil) \\
= \mathit{split}(\mathsf{T}, V_1, V_2, \lceil\texttt{let}\ \langle x_1\rangle = \langle e\rangle\ \texttt{in}\ (\langle a_1\rangle \rhd \lambda\langle x_2\rangle \to \langle a_2\rangle)\rceil)
\end{array}
\tag{4}
$$

$$
\begin{array}{l}
\mathit{split}(\mathsf{T}, V_1, V_2, \lceil\langle a_1\rangle \rhd \lambda\langle x_2\rangle \to \langle a_2\rangle\rceil) \\
=\ \ \mathit{let}\ (i, W_1, W_2, B, r_1, r_2) = \mathit{split}(\mathsf{T}, \lceil()\rceil, \lceil()\rceil, a_1)\ \mathit{in} \\
\quad\ \mathit{if}\ i = 1\ \mathit{then} \\
\qquad \mathit{let}\ (i', W_1', W_2', B', r_1', r_2') = \mathit{split}(\mathsf{T}, \lceil(\langle x_2\rangle, (\langle W_1\rangle, \langle V_1\rangle))\rceil, \lceil(\langle W_2\rangle, \langle V_2\rangle)\rceil, a_2) \\
\qquad \mathit{in}\ (i', W_1', W_2', \lceil\langle B'\rangle; \langle B\rangle\rceil, \langle r_1\rangle \rhd \lambda(\langle x_2\rangle, \langle W_1\rangle) \to \langle r_1'\rangle, \langle r_2\rangle \rhd \lambda\langle W_2\rangle \to \langle r_2'\rangle) \\
\quad\ \mathit{else\ if}\ i = 2\ \mathit{then}\ \ldots
\end{array}
\tag{5}
$$

$$
\begin{array}{l}
\mathit{split}(\mathsf{T}, V_1, V_2, \lceil\langle a_1\rangle \rhd \lambda\langle x\rangle \to \langle a_2\rangle\rceil) \\
=\ \ \mathit{let}\ M = \mathit{least\text{-}effect}(a_1)\ \mathit{in} \\
\quad\ \mathit{if}\ M \subseteq \mathsf{T}\ \mathit{then} \\
\qquad \mathit{let}\ (i, W_1, W_2, B, r_1, r_2) = \mathit{split}(\mathsf{T}, \lceil(\langle x\rangle, \langle V_1\rangle)\rceil, V_2, a_2) \\
\qquad \mathit{in}\ (i, W_1, W_2, B, \lceil\langle a_1\rangle \rhd \lambda\langle x\rangle \to \langle r_1\rangle\rceil, r_2) \\
\quad\ \mathit{else\ if}\ \mathsf{T} \cap M = \emptyset\ \mathit{then} \\
\qquad \mathit{let}\ (i, W_1, W_2, B, r_1, r_2) = \mathit{split}(\mathsf{T}, V_1, \lceil(\langle x\rangle, \langle V_2\rangle)\rceil, a_2) \\
\qquad \mathit{in}\ (i, W_1, W_2, B, r_1, \lceil\langle a_1\rangle \rhd \lambda\langle x\rangle \to \langle r_2\rangle\rceil)
\end{array}
\tag{6}
$$

$$
\begin{array}{l}
\mathit{split}(\mathsf{T}, V_1, V_2, a) \\
=\ \ \mathit{let}\ M = \mathit{least\text{-}effect}(a)\ \mathit{in} \\
\quad\ \mathit{if}\ M \subseteq \mathsf{T}\ \mathit{then} \\
\qquad (1, V_1, V_2, \varepsilon, \lceil\langle a\rangle \rhd \lambda z \to \texttt{result}\ (z, \langle V_1\rangle)\rceil, \lceil\texttt{result}\ \langle V_2\rangle\rceil) \\
\quad\ \mathit{else\ if}\ \mathsf{T} \cap M = \emptyset\ \mathit{then} \\
\qquad (2, V_1, V_2, \varepsilon, \lceil\texttt{result}\ \langle V_1\rangle\rceil, \lceil\langle a\rangle \rhd \lambda z \to \texttt{result}\ (z, \langle V_2\rangle)\rceil)
\end{array}
\tag{7}
$$

Figure 3: Rules for *split*

It runs its argument action a independently (possibly concurrently) from the following computation by ignoring the final state of its argument:

```
interleave a
 = λs -> let (x', s') = a s in (x', s)
```

The duplication of the state s in `interleave` is potentially dangerous. In `interleave a ▷ λv -> a'` the action a as well as the action a' can read and write the same variables. However, if a only operates on terminated references and includes their terminators the subsequent action a' cannot have access to them and hence cannot interfere with the interleaved action. Thus interleaving computations on terminated references is safe.

Intertwined in a single action the slices communicated via variable bindings. Fortunately, *split* collects the bound variables of the slices in V_1 and V_2. The necessary feedback operation from one slice to the other can be achieved with the combinator `fixST` ([LP94]), defined as:

```
fixST :: (v -> ST s v) -> ST s v
fixST f = λs -> let (x', s') = f x' s in (x', s')
```

Using `fixST` and `interleave` the main transformation is specified as follows

$$
\begin{aligned}
&\mathit{extract}(\mathsf{T}, a)\\
&= \ \mathit{let}\ (i, V_1, V_2, B, r_1, r_2) = \mathit{split}(\mathsf{T}, \lceil() \rceil, \lceil() \rceil, a)\ \mathit{in}\\
&\quad\ \mathit{if}\ i = 1\ \mathit{then}\\
&\qquad \lceil \texttt{fixST}(\ \lambda z_0 \rightarrow\\
&\qquad\qquad \texttt{let}\ (z_1, \langle V_1 \rangle, \langle V_2 \rangle) = z_0; \langle B \rangle\ \texttt{in}\\
&\qquad\qquad \texttt{interleave}\ \langle r_1 \rangle \rhd \lambda(z_1, \langle V_1 \rangle) \rightarrow\\
&\qquad\qquad \langle r_2 \rangle \rhd \lambda\langle V_2 \rangle \rightarrow\\
&\qquad\qquad \texttt{result}\ (z_1, \langle V_1 \rangle, \langle V_2 \rangle))\\
&\qquad \rhd \lambda(z_1, \langle V_1 \rangle, \langle V_2 \rangle) \rightarrow \texttt{result}\ z_1 \rceil\\
&\quad\ \mathit{else}\\
&\qquad \lceil \texttt{fixST}(\ \lambda z_0 \rightarrow\\
&\qquad\qquad \texttt{let}\ (z_2, \langle V_1 \rangle, \langle V_2 \rangle) = z_0; \langle B \rangle\ \texttt{in}\\
&\qquad\qquad \texttt{interleave}\ \langle r_1 \rangle \rhd \lambda\langle V_1 \rangle \rightarrow\\
&\qquad\qquad \langle r_2 \rangle \rhd \lambda(z_2, \langle V_2 \rangle) \rightarrow\\
&\qquad\qquad \texttt{result}\ (z_2, \langle V_1 \rangle, \langle V_2 \rangle))\\
&\qquad \rhd \lambda(z_2, \langle V_1 \rangle, \langle V_2 \rangle) \rightarrow \texttt{result}\ z_2 \rceil
\end{aligned}
$$

The application of `fixST` serves to equate the argument of the final `result` $(z_1, \langle V_1 \rangle, \langle V_2 \rangle)$ with the argument z_0 of its argument function. Since the final result contains the values of all bound variables, these values are available in the whole body of `fixST`'s argument function: the binders B, r_1, and r_2. The transformed expression depends crucially on the irrefutability of the pattern binding $(z_1, \langle V_1 \rangle, \langle V_2 \rangle) = z_0$. Otherwise the required cyclic bindings cannot be established.

6.1 Example: Lazy File Read

The motivating example of [PW93, LP94] for `interleave` is the implementation of lazy file read. The state monad cannot only be used to ensure correct sequencing of actions on references, but it can sequence I/O actions as well. Usually a more strict version `thenIO` of the *bind* operator is employed for I/O. `thenIO` is strict in the propagated state and forces the immediate execution of its argument actions, since there is no point in delaying I/O actions.

A typical subset of primitive I/O actions is `openFile :: String -> ST s (Fildes t s)` (opens a file and returns a file descriptor), `eof :: Fildes t s -> ST s Bool` (tests for end of file), `readChar :: Fildes t s -> ST s Char` (reads a character), and `closeFile :: Fildes t s -> ST s ()` (closes a file). An action which reads the contents of a file into a list of characters can be written as follows.

```
readFile :: String -> ST s String
readFile fname = openFile fname ▷ λfd ->
    let readContents
         = eof fd ▷ λdone ->
           if done then
                closeFile fd `thenIO` λ_ ->
                result []
           else readChar fd  ▷ λch ->
                readContents ▷ λrest ->
                result (ch:rest)
    in
        readContents ▷ λcont ->
        result cont
```

In the program fragment `readFile f ▷ λchs -> next_action`, the first read or write action in `next_action` requires the complete execution of `readFile f` due to the sequentialization induced by *bind*. That is, the entire file `f` is read before any other I/O action can take place. Reading the contents of `f` lazily on demand would be much more efficient.

Let us introduce a terminator `termFile` of type `∀t. Fildes t s -> ST s ()`, which is strict in the file descriptor of type `Fildes` and in the state, and place it right in front of the final `result cont`, as in:

```
...
        readContents ▷ λcont ->
        termFile fd  ▷ λ_ ->
        result cont
```

Let $\underline{t}$ be the terminator type of `fd` (from the application `termFile fd`). Applying the *extract* transformation with $\mathsf{T} = \{\underline{t}\}$ to the body of `readFile` yields:

```
readFile fname =
  fixST (λz0 ->
        let (cont, fd) = z0 in
```

```
let readContents = ...
in  interleave (
        openFile fname ▷ λfd ->
        readContents   ▷ λcont ->
        termFile fd    ▷ λ_ ->
        result cont
    ) ▷ λcont ->
    result (cont, fd)
) ▷ λ(cont, fd) ->
result cont
```

7 Extracting Unterminated Computations

Computations can be extracted from a state action even if they are not terminated. However, `interleave` cannot be used to parallelize them, since subsequent actions depend on the outcome and effects of both slices. Therefore, we introduce a combinator `fork` which accepts two state actions and executes them independently.

```
fork :: ST s v -> ST s w -> ST s (v, w)
fork act1 act2 s
      = let (x1, s1) = act1 s
            (x2, s2) = act2 s
        in ((x1, x2), mergeStates s1 s2)
```

The function `mergeStates` has type `a -> a -> a` and is strict in both arguments, such that a demand on the output state of `fork act1 act2` causes a demand on the output states `s1` and `s2` of both `act1` and `act2` which in turn forces their execution. The `fork` operator as defined above enables the *interleaved* execution of its argument actions `act1` and `act2`. As in the case of `interleave`, the use of `fork` is potentially dangerous. It can only be guaranteed to be correct if the argument actions do not interfere, *i.e.* `act1` does not read references that are written to by `act2`, `act1` does not write references that are read or written by `act2`, and vice versa. Of course, our transformation only introduces safe `fork`s since it is only applied to slices which operate on disjoint sets of references.

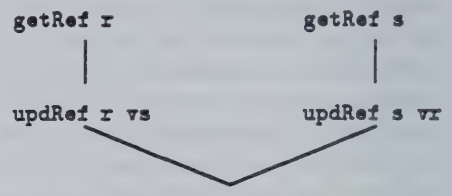
In a parallel implementation the `let` in the definiton of `fork` could be replaced by a `letpar` construct which starts the evaluation of its binders in parallel.

Notice that `fork` as defined above behaves different to the `fork` operator in recent work of Jones and Hudak [JH93]. Their `fork` only interleaves actions if they make use of explicit communication primitives.

The transformation becomes pleasingly symmetric using `fork`, as shown in Fig. 4. As an example for an application of *extract'* consider the function `swap` which creates an action which exchanges the contents of two references.

```
swap :: Ref! t1 s v -> Ref! t2 s v -> ST s ()
swap r s = getRef r   ▷ λv -> getRef s ▷ λw ->
           updRef r w ▷ λ_ -> updRef s v
```

The state monad sequentializes `swap` too much (albeit on a trivial scale): reading the references can be done in parallel, as well as updating them, as long as no reference is overwritten before it is read out. The most liberal order looks like this:



Application of *extract'* to the body of `swap` yields (after some simplification):

```
pswap r s =
 fixST (λz0 ->
        let (vr, (z2, vs)) = z0 in
        fork
           (getRef r    ▷ λvr ->
            updRef r vs ▷ λ_ ->
            result vr)
           (getRef s    ▷ λvs ->
            updRef s vr ▷ λz2 ->
            result (z2, vs)))
 ▷ λ(_, (z2, _)) -> result z2
```

While the above action `pswap` works correctly even if its arguments reference the same mutable variable, this is not true in general. As an example consider the following action `munge` which performs the assignment $(r, s) := (r + s, r - s)$.

```
munge :: Ref! t1 s v -> Ref! t2 s v -> ST s ()
munge r s = getRef r        ▷
λv -> getRef s ▷ λw ->
            updRef r (v+w) ▷
λ_ -> updRef s (v-w)
```

`munge` can only be parallelized along the lines of `swap` if `r` and `s` cannot be aliases for the same variable. Fortunately, the terminator type of the involved references indicates aliasing: when `r` and `s` may be aliases of one another their terminator types are equal.

It turns out that terminator types can be kept different by adding inequation constraints of the form $\alpha \neq \beta$ to the types of state transformers. For example, the type of a parallelized `munge` would be

```
pmunge :: Ref! t1 s v -> Ref! t2 s v
                      -> ST^{t1, t2} s ()
          if t1 /= t2
```

With this type we could split `pmunge` as follows:

$$
\begin{aligned}
&\mathit{extract}'(T, a)\\
&= \ \mathit{let}\ (i, V_1, V_2, B, r_1, r_2) = \mathit{split}(T, \lceil() \rceil, \lceil() \rceil, a)\ \mathit{in}\\
&\quad\ \mathit{if}\ i = 1\ \mathit{then}\\
&\qquad \lceil \mathtt{fixST}(\lambda z_0 \rightarrow\ \mathtt{let}\ ((z_1, \langle V_1 \rangle), \langle V_2 \rangle) = z_0; \langle B \rangle\ \mathtt{in}\\
&\qquad\qquad\qquad\qquad \mathtt{fork}\ \langle r_1 \rangle\ \langle r_2 \rangle)\\
&\qquad \rhd \lambda((z_1, \langle V_1 \rangle), \langle V_2 \rangle) \rightarrow \mathtt{result}\ z_1 \rceil\\
&\quad\ \mathit{else}\\
&\qquad \lceil \mathtt{fixST}(\lambda z_0 \rightarrow\ \mathtt{let}\ (\langle V_1 \rangle, (z_2, \langle V_2 \rangle)) = z_0; \langle B \rangle\ \mathtt{in}\\
&\qquad\qquad\qquad\qquad \mathtt{fork}\ \langle r_1 \rangle\ \langle r_2 \rangle)\\
&\qquad \rhd \lambda(\langle V_1 \rangle, (z_2, \langle V_2 \rangle)) \rightarrow \mathtt{result}\ z_2 \rceil
\end{aligned}
$$

Figure 4: Extraction for unterminated references

```
pmunge r s =
 fixST (λz0 ->
        let (v, (z2, w)) = z0 in
        fork
           (getRef r       ▷ λv ->
            updRef r (v+w) ▷ λ_ ->
            result v)
           (getRef s       ▷ λw ->
            updRef s (v-w) ▷ λz2 ->
            result (z2, w)))
 ▷ λ(_, (z2, _)) -> result z2
```

Another modification of the type checker helps to enforce the inequation constraints: they must be checked whenever the unification algorithm extends the current substitution.

8 Synchronization

However satisfactory the transformation works in the preceding example, it is not yet complete in general. There is still a synchronization problem: what happens if an action in the extracted slice refers to a reference manipulated by the remaining slice? An example is the action `swap r s` where `r` "belongs" to one slice and `s` to the other. `swap r s` cannot simply be assigned to one of the slices since doing so can introduce side effects. The solution is to synchronize the slices such that `swap r s` can only be executed when all preceding computations that are assigned to the other slice are done and that the other slice can only continue after `swap r s` has been executed. To achieve the synchronization, a simple handshaking protocol can be implemented by introducing new λ-abstractions and using the functions `wait :: SyncToken -> ST s ()`, which is strict in its `SyncToken` argument and in the state, and `ready :: ST s SyncToken`, which is strict in the state `s`. `SyncToken` may be any non-empty type, for example `type SyncToken = ()` will do.

The missing case in the definition of rule (6) for *split* is specified in Fig. 5. Before any read or write operation in a_1 in the first slice can be executed, `wait` $\hat{y}$ has to receive a `SyncToken` through the variable $\hat{y}$. Then it performs a_1, signals `ready` through variable $\hat{x}$, and continues with r_1. Likewise the second slice first signals on $\hat{y}$ that it is ready to perform r_2, but r_2 cannot proceed before `wait` $\hat{x}$ has received its `SyncToken` through $\hat{x}$. Depending on the context of the slices and the least effect of r_1 and r_2, part of the protocol may be omitted.

The same synchronization problem occurs if two references which are manipulated by different slices may be aliases of one another. Two references are not aliased if their terminator types cannot be equal. In the case of references which may be aliased the original sequence of their update actions must be retained using the mechanism outlined above.

9 Related Work

Following Moggi [Mog89], Wadler [Wad90, Wad92] advocates the use of monads to express state-based computations in functional programming. Peyton Jones and Wadler [PW93] have used the state monad to ensure proper sequencing of I/O operations and assignment operations on references.

Launchbury describes a lazy implementation of mutable arrays and variables in terms of a sequencing monad [Lau93]. The independence of different incarnations (and thus referential transparency) of the sequencing monad is ensured by assigning each sequencing thread a unique identity. Each operation must first check the identity in order to guarantee absence of side-effects. This expensive check is not necessary using our approach where the type checker ensures matching identities. More recent work of Launchbury and Peyton Jones [LP94] also avoids the

$$
\begin{array}{l}
\mathit{split}(\mathsf{T}, V_1, V_2, \lceil \langle a_1 \rangle \rhd \lambda\langle z \rangle \rightarrow \langle a_2 \rangle \rceil) \\
= \ \mathit{let}\ \mathsf{M} = \mathit{least\text{-}effect}(a_1)\ \mathit{in} \\
\quad \mathit{if}\ \mathsf{T} \cap \mathsf{M} \neq \emptyset\ \mathit{and}\ \mathsf{M} \not\subseteq \mathsf{T}\ \mathit{then} \\
\qquad \mathit{let}\ \hat{\mathsf{x}}\ \mathit{and}\ \hat{\mathsf{y}}\ \mathit{be\ fresh\ variables\ in} \\
\qquad \mathit{let}\ (\mathsf{i}, W_1, W_2, \tau_1, \tau_2) = \mathit{split}(t_1, \lceil (\langle z \rangle, (\hat{\mathsf{x}}, \langle V_1 \rangle)) \rceil, \lceil (\hat{\mathsf{y}}, \langle V_2 \rangle) \rceil, a_2) \\
\qquad \mathit{in}\ (\mathsf{i}, W_1, W_2, \\
\qquad\quad \lceil \mathtt{wait}\ \hat{\mathsf{y}} \rhd \lambda() \rightarrow \langle a_1 \rangle \rhd \lambda\langle z \rangle \rightarrow \mathtt{ready} \rhd \lambda\hat{\mathsf{x}} \rightarrow \langle \tau_1 \rangle \rceil, \\
\qquad\quad \lceil \mathtt{ready} \rhd \lambda\hat{\mathsf{y}} \rightarrow \mathtt{wait}\ \hat{\mathsf{x}} \rhd \lambda() \rightarrow \langle \tau_2 \rangle \rceil) \\
\quad \mathit{else}\ \ldots
\end{array}
$$

Figure 5: Synchronization for *split* .

check and achieves the encapsulation (as stated under 1. above) by introducing a constant with a rank-2 polymorphic type.

Hudak [Hud93] describes several methods for defining mutable abstract datatypes which encapsulate single-threaded computations and which are amenable to implementation with assignments. He generates the specification of a mutable abstract datatype from the specification of an immutable one if it obeys a linearity condition. He proposes first-class references but does not address the problem of escaping references.

Jones and Hudak [JH93] propose a monadic framework for parallel functional programming. The source of parallelism in their work is an unsafe function `fork` where every use must be proved not to introduce side effects. In our work `fork` is introduced by a transformation and is thus correct once the correctness of the transformation is established.

Chen and Odersky [CO93] describe a stratified type system for $\lambda_{\mathbf{var}}$ [ORH93], a lambda calculus extended by imperative features. The system has a type reconstruction algorithm. It is not capable of handling nested state transformers which compute state transformers themselves. The type system proposed here provides that capability. Furthermore, $\lambda_{\mathbf{var}}$ is much more strict than our intended semantics. In $\lambda_{\mathbf{var}}$, a state-based computation can only return a value after all operations (assignments and reads) on the state are completed. State transformers can return partial results since they implement lazy state and can defer some operations on the state.

Swarup, Reddy, and Ireland [SRI91] have defined a calculus which extends the lambda calculus by reference variables and operations on them. The absence of side-effects in the calculus relies on a stratified type system which distinguishes pure expressions, observers, and mutators. The stratification makes sure that no references can escape from their scope, but also that no observers can be constructed imperatively (which is possible in our work). Since the programming style in their calculus is strongly influenced by continuation-passing style, it is questionable whether parallelization can be achieved solely by program transformation, as suggested here.

Smetsers et al. [SBvP93] describe a uniqueness type system for term graph rewrite systems. If a node of a redex can be assigned a unique type the corresponding node can be updated destructively, *i.e.* it can be reused in the contractum. They achieve a similar effect, the ability to do computations with assignments, by quite different means.

10 Conclusion

The paper presents further evidence for the power of the monadic approach to integrate first-class references to mutable objects in purely functional programming languages. The concepts of upward and downward abstract types allow a high degree of control over the lifetime of references. References are safely confined to their creating thread, while "freezing" a mutable object to an immutable one means to let a reference escape in a controlled way. Parallelism can be introduced in the sequential state monad by an effect-directed program transformation which automatically exposes implicit parallelism. It seems unlikely that a similar transformation is possible for continuation-based approaches.

Downward and upward abstract types can also be used to ensure encapsulation and termination of references for other mutable abstract datatypes, for example the direct style and the continuation-passing style mutable abstract datatypes of Hudak [Hud93].

The automatic extraction of independent parts of state threads as shown in Sections 5.2 and 7 does not depend on our specific encapsulation mechanism, it works as well with the encapsulation proposed by

Launchbury and Peyton Jones [LP94].

A Operational Semantics

The intended setting for the development in the main text is an enriched lazy λ-calculus. Its syntax is given by

$$\Lambda ::= \mathsf{K} \mid \mathsf{X} \mid (\lambda \mathsf{X}.\Lambda) \mid (\Lambda\ \Lambda) \mid \texttt{let}\ \mathsf{X} = \Lambda\ \texttt{in}\ \Lambda$$

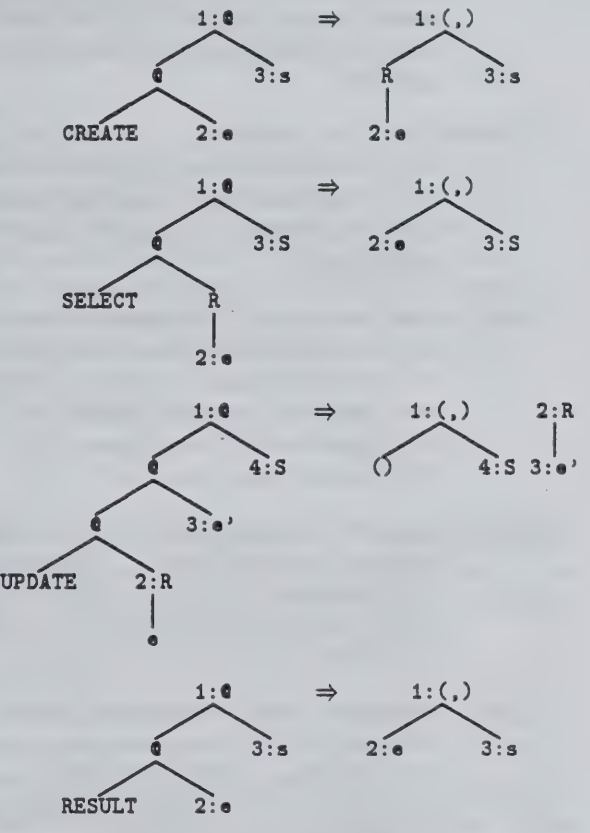
where X is an infinite set of variables and K is a set of constant symbols each with a specific arity (a natural number). The usual conventions for omitting parentheses apply throughout (application associates to the left and the scope of an abstraction extends as far to the right as possible).

The operational semantics is defined by graph rewriting as originally proposed by Wadsworth [Wad71], see also [Pey87]. Graphs are rewritten according to the functional strategy [KSvP93] which enriches priority-based term graph rewriting with demand driven rewriting due to pattern matching. Graph nodes are labelled with constants, variables, λx, or @ (application) and each node has a list of successors. The number of successors corresponds to the arity of the label (the arities of variable nodes, λx nodes, and @ nodes is 0, 1, and 2, respectively).

An expression $e \in \Lambda$ is translated into a rooted graph as follows: a constant k or a variable x results in a node labelled k or x, an abstraction $\lambda x.e$ results in a (root) node labelled λx with the graph for e as the only successor, an application $(e_1\ e_2)$ results in an application node (the root) with the graphs for e_1 and e_2 as successors, the graph for $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ consists of the graphs for e_1 and e_2 where the root is the root of e_2 and every link to a free occurrence of x in e_1 and e_2 replaced by a link to the root of e_1.

The functional strategy starts at the root of a graph searching for a redex. A redex is a subgraph, either a β-redex $(\lambda x.e_2)e_1$ (an application of an abstraction) or a rule-redex $\mathsf{f}\ d_1 \ldots d_n$ (an n-fold applcation of a nullary constant to arguments). If neither is found the graph is not reduced any further. A β-redex is replaced by a fresh copy of the body e_2 with every link to a free occurrence of x replaced by a link to e_1. In case of a rule-redex the rules for f are considered in textual order. In each rule the argument patterns on the left hand side are considered from left to right. The nodes of each argument pattern are considered recursively. Recursion stops if the considered node is a variable. If it is labelled with a constant, the rewriting process is applied recursively to the corresponding argument node of the redex. Then the successors of the argument node (if any) are considered recursively.

We assume that there are distinct constants CREATE, SELECT, UPDATE, RESULT ∈ K with associated rules to implement the operations on references. The rules will be described graphically, with the convention that special symbols @ (label of an application node), (,) (label for a pair constructor), () (the unit tuple) and symbols written in upper case letters are constant labels, whereas lower case letters denote arbitrary nodes in the graph. When some node from the left hand side of a rewriting rule is reused on the right hand side it is labelled with a number as in 1:x. If the label of such a node changes it is overwritten in the rewriting step. All other nodes are created afresh.



Notice that the state argument in the rules for `select` and `update` is a data constructor, i.e. a constant which must be matched explicitly. Hence reduction of one of these rules forces the evaluation of the state. On the contrary, reduction of the rules for `create` and `result` do *not* force the evaluation of the state.

References

[CO93] Kung Chen and Martin Odersky. A type system for a lambda calculus with assignment. Research Report YALEU/DCS/RR-963, Department of Computer Science, Yale University, New Haven, Connecticut, May 1993.

[Has92] Report on the programming language Haskell, a non-strict, purely functional language, version 1.2. *SIGPLAN Notices*, 27(5):R1–R164, May 1992.

[Hud93] Paul Hudak. Mutable abstract datatypes —or— how to have your state and munge it too. Research Report YALEU/DCS/RR-914, Yale University, Department of Computer Science, New Haven, CT, December 1993. (Revised May 1993).

[JH93] Mark P. Jones and Paul Hudak. Implicit and explicit parallel programming in Haskell. Research Report YALEU/DCS/RR-982, Yale University, New Haven, CT 06520-2158, August 1993.

[KSvP93] P. W. M. Koopman, J. E. W. Smetsers, M. C. J. D. van Eekelen, and M. J. Plasmeijer. *Graph Rewriting Using the Annotated Functional Strategy*, chapter 23. John Wiley & Sons, 1993.

[Lau93] John Launchbury. Lazy imperative programming. In Paul Hudak, editor, *SIPL '93, ACM SIGPLAN Workshop on State in Programming Languages*, number YALEU/DCS/RR-968, pages 46–56, New Haven, CT, June 1993. Copenhagen, Denmark.

[LG88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, pages 47–57, San Diego, CA, January 1988.

[LO92] Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *Proc. ACM SIGPLAN Workshop on ML and its Applications*, pages 78–91, San Francisco, CA, June 1992.

[LP94] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Proc. of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 24–35, Orlando, Fla, USA, June 1994. ACM Press. ACM SIPLAN Notices, v29, 6.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proc. of the 4rd Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, CA, June 1989. IEEE Computer Society Press.

[MP88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[ORH93] Martin Odersky, Daniel Rabin, and Paul Hudak. Call-by-name, assignment, and the lambda calculus. In POPL1993 [POP93], pages 43–57.

[Pey87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[POP93] *Proc. 20th ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993. ACM Press.

[PW93] Simon L. Peyton Jones and Philip L. Wadler. Imperative functional programming. In POPL1993 [POP93], pages 71–84.

[SBvP93] Sjaak Smetsers, Erik Barendsen, Marko van Eekelen, and Rinus Plasmeijer. Guaranteeing safe destructive updates through a type system with uniqueness information for graphs. Technical Report 93-4, Department of Computer Science, University of Nijmegen, 1993.

[SRI91] Vipin Swarup, Uday S. Reddy, and Evan Ireland. Assignments for applicative languages. In John Hughes, editor, *Proc. Functional Programming Languages and Computer Architecture 1991*, pages 192–214, Cambridge, MA, 1991. Springer-Verlag. LNCS 523.

[Thi93] Peter Thiemann. Safe sequencing of assignments in purely functional programming languages. Technical Report WSI-93-16, Wilhelm-Schickard-Institut, Tübingen, Germany, November 1993.

[Wad71] Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Programming Research Group, Oxford University, 1971.

[Wad90] Philip L. Wadler. Comprehending monads. In *Proc. Conference on Lisp and Functional Programming*, pages 61–78, Nice, France, 1990. ACM.

[Wad92] Philip L. Wadler. The essence of functional programming. In *Proc. 19th ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992.

Mutable Data Structures and Composable References in a Pure Functional Language

Koji Kagawa
Research Institute for Mathematical Sciences
Kyoto University
Kyoto 606-01, Japan
E-mail: kagawa@kurims.kyoto-u.ac.jp

Abstract

We propose composable references as tools to handle mutable data structures in monadic-style functional programs. They enable us to express mutable data structures efficiently, for example, to avoid unnecessary indirections and to avoid copying. A composable reference is a reference which is parameterized over the type of the underlying mutable data structure used as the state and intuitively is a location of a field (or more generally, a substructure) relative to the parameterized state. Composable references can be composed as functions. Composable references make stateful programs concise by allowing mutable data structures to be passed implicitly.

1 Introduction

The notion of multiple mutable objects or references was a problematic issue in purely functional languages such as Haskell, when the monad of state transformers [Wad90, Wad92] was introduced to express stateful computations, that is, to enable in-place updating while retaining referential transparency. Some solutions have been proposed to this problem and some of them, notably that of Launchbury and Peyton Jones [LP94], have been already incorporated into existing implementations. In their proposal, however, only two data structures (`MutVar` (reference cells) and `MutArr` (mutable arrays)) are essentially mutable.

Based on these works, in this paper, we propose a way to deal with general data structures such as tuples and lists with mutable components. These are different from those data structures whose fields are of `MutVar` type in the sense that we need indirections in the latter as is shown in Figure 2. Mutable objects should have representations as efficient as those used in conventional imperative languages (such as C). Caml Light [Ler93] has already such mutable data structures, but it is a strict, impure language and referential transparency does not need to be maintained.

A composable reference intuitively stands for a location of a mutable field (or a substructure) relative to the mutable data structure such as tuples and cons cells and is parameterized over the type of the mutable data structure. We use composable references in order to access fields (substructures) of mutable data structures. State transformers are written relatively to mutable data structures used as local states and then are composed with composable references. We can also compose two parameterized references when the intermediate mutable data structure contains another mutable object as its substructure as we can compose two functions $f :: a \to b$ and $g :: b \to c$ using function composition ($\circ$). Then we can construct stateful programs hierarchically.

Composable references allow substructures of mutable data structures to be passed to stateful functions which expect the type of these substructures without creating actual copies of these substructures. This means that composable references can play a role which cast operators play in e.g. C in the sense that both are used to coerce mutable objects into another data type in order to access their substructures without actually creating their copies.

Though we discuss in the context of a lazy language with overloading, there appears to be no essential dependency upon evaluation strategies. The technique used in this paper should be applicable both to strict languages such as ML and to lazy languages such as Haskell. Monads (and therefore the work presented here) can be useful also for strict languages. The reason is as follows. First, we do not need weak type variables as is pointed out in [PJW93]. Second, we have more opportunities to find type errors; we can not place expressions which have no side effect where those with side effect are expected. And last, we can distinguish the pure functional parts from the imperative parts.

In this paper, we assume some familiarity with the Haskell syntax. In the remainder of this section, we review related works.

1.1 Monadic operators

Monads provide a uniform framework for various forms of sequential computations (i.e. those with *side effect*) and have been popularized in functional programming community. In this paper, we do not get into details and only introduce some *monadic* operators which are used to express stateful computations. Figure 1 explains the monad of state transformers.

For examples and motivations, see [Mog89, Wad90, Wad92]. The two monads used here — the monad of state transformers and the monad of state readers — are in fact families of monads which differ only in the type of the state. *In-place updating* is crucial in stateful computations. The designer of primitive stateful operators must carefully hide states from programmers so that they can not duplicate states. The monad of state transformers is considered to be a nice abstraction for this purpose. The implementation of the `ST` and `SR` monad is hidden so that only predefined operators are applied to states and that programmers never duplicate pointers to data structures which represent states. Then the designer must implement carefully read and write primitives.

1.2 Launchbury and Peyton Jones

When monads were introduced to functional programming, it was problematic whether it was possible to manipulate more than one piece of mutable object. In order to solve this problem, Launchbury and Peyton Jones proposed the primitive operator `newVar` [LP94] which creates new reference cells (of type `MutVar`):

```
newVar   :: a -> ST s (MutVar s a)
```

and associated operators `readVar` and `writeVar`:

```
readVar  :: MutVar s a -> ST s a
writeVar :: MutVar s a -> a -> ST s ()
```

Here, `MutVar` is parameterized over the type of the state. This parameterized type of the state is used in a special typing rule which prevents references from escaping their scopes made by `runST`.

```
runST :: ∀a. (∀s. ST s a) -> a
```

```
-- the monad of state transformers
data ST s a = MkST (s -> (a, s))

returnST :: a -> ST s a
returnST a = MkST (\ s -> (a, s))

thenST :: ST s a -> (a -> ST s b) -> ST s b
(MkST m) `thenST` k = MkST (\ s0 -> let (a, s1) = m s0; MkST m' = k a
                                    in m' s1)

-- the monad of state readers.
data SR s a = MkSR (s -> a)

returnSR :: a -> SR s a
returnSR a = MkSR (\ s -> a)

thenSR :: SR s a -> (a -> SR s b) -> SR s b
(MkSR m) `thenSR` k = MkSR (\ s -> let a = m s; MkSR m'= k a
                                   in m' s)
```

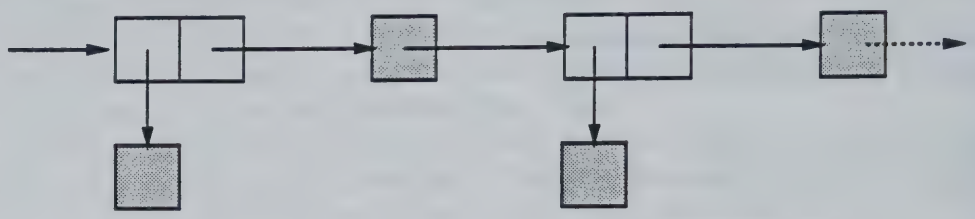
Figure 1: Monadic Operators

When **runST** is applied to a state transformer, it passes an initial "empty" state to the state transformer and then extract the result. In this way **runST** makes it possible to extract a "pure" value from a stateful computation. The typing rule above is necessary because otherwise a reference (**MutVar**) created in one "state thread" could be used in another thread by mistake and because detecting errors caused by such *cross* pointers in runtime would be expensive.

There are, however, only two data structures which are essentially mutable (mutable cells and mutable arrays). There is no other primitive mutable structures such as pairs, lists and records. These are different from data structures whose fields are of **MutVar** type, for we need indirect cells in the latter. This situation is illustrated in Figure 2 which compares two representations of mutable lists. Using reference cells, the tail part of a cons cell can not directly point to another cons cell. This is problematic when we need an efficient representation of mutable graph structures. What is lacking seems to be an appropriate method to handle such primitive mutable data structures.

1.3 Concurrent Clean

In [AvGP93], Achten, van Groningen and Plasmeijer proposed *Unique types* to ensure uniqueness of data structures and thus to make in-place updating possible. Though this is an approach quite different from using monads, unique types in Concurrent Clean are another motivation of this work. Because data structures which represent states are passed around explicitly, we can use a hierarchy of states to write modular programs and to separate non-interfering parts of a program so that they can run in parallel. On the other hand, there is no notion of references and this makes mutable graph structures with shared nodes difficult to handle. We must imitate mutable graph structures by using, for example, an integer as an identity of a node, which is, at least, inefficient.



The representation of mutable lists using reference cells

The desirable representation of mutable lists

Figure 2: Comparison (The shaded parts are mutable.)

2 Composable references

It is unrealistic to define primitive read/write operators for each field of each data type in order to handle mutable data structures such as tuples, lists and records. We need a set of primitives which is as small as possible. We will use composable references in order to make use of these primitives to handle general mutable data structures.

2.1 Operators for composable references

As is said in the introduction, a composable reference intuitively means a location of a field or a substructure relative to the data structure such as tuples and records used as a state. In the following, we will explain how they interact with state transformers and how they should be implemented.

We think of composable references (`CR`) as an abstract data type with the following operations:

```
(-!>) :: CR s t -> ST t x -> ST s x
(-?>) :: CR s t -> SR t x -> SR s x
(-*>) :: CR s t -> CR t u -> CR s u
```

We can think of composable references as generators of monad morphisms between state transformers. Monad morphisms are functions between two monads with some pleasant properties. For the definition and examples, please refer to e.g. [Wad90]. We assume these operators bind tighter than `'then*'` and associate to left. We use these operators to compose stateful programs. The meanings of these operator are as follows. When `CR s t`, which means a relative location of a substructure of type `t` in `s`, and `ST t x`, which is a state transformer for type `t`, are combined by `(-!>)`, the result is a state transformer for type `s` which, however, only modifies the substructure of type `t` pointed by the composable reference. The result of the composition of `CR s t` and `CR t u` by `(-*>)` is the relative position of the substructure of type `u` with respect to the structure of type `s`.

This means that state transformers are written with respect to local states and that we pass data structures which are used as local states implicitly to state transformers. It seems that the implicit style is generally preferred because it allows us to write concise and modular codes as experiences in object-oriented programming suggest. In addition, we can separate non-interfering

```
data CR s t = MkCR (s -> (t, t -> s))

(MkCR r) -!> (MkST st) = MkST (\ s -> let (t, t2s) = r s
                                          (x, t')  = st t
                                      in (x, t2s t'))

(MkCR r) -?> (MkSR sr) = MkSR (\ s -> let (t, _) = r s
                                      in sr t)

(MkCR r) -*> (MkCR q)  = MkCR (\ s -> let (t, t2s) = r s
                                          (u, u2t) = q t
                                      in (u, t2s.u2t))

idR :: CR a a
idR =  MkCR (\ a -> (a, id))
```

Figure 3: An implementation of CR

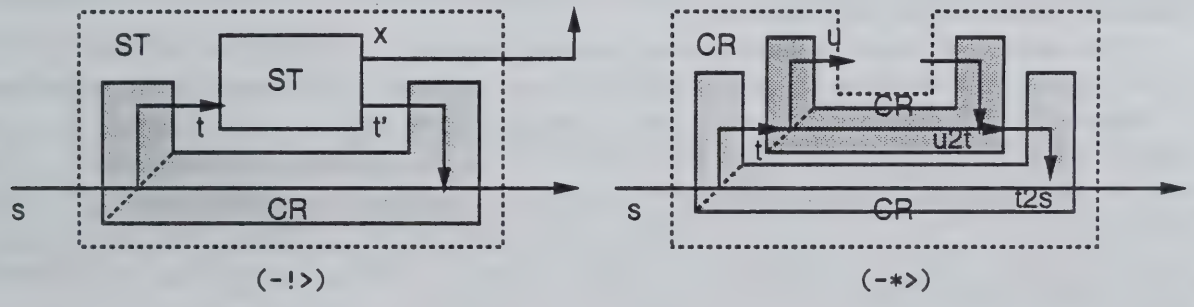


Figure 4: A diagram notation of CR

parts of a stateful program as in Concurrent Clean. This might be useful for parallelizing execution of stateful programs.

We will use the code of Figure 3 to specify composable references and various operations upon composable references and state transformers. When a composable reference (of type `CR s t`) is applied to a data structure (of type `s`), the result is a pair of an extracted substructure (of type `t`) and a *put-back* function of type `t -> s`. This extracted state is passed to a state transformer (`(-!>)`) or another composable reference (`(-*>)`). If we use the diagram notation of [LP94], a composable reference and associated operators would be expressed as in Figure 4.

2.2 Composable references for field access

As state transformers can be implemented so that they can do in-place updating by restricting their expressiveness, composable references can be implemented so that they do not need to create actual copies of substructures as far as the layout of data structures allows. We will consider only such composable references in the following. Let us consider the following composable references

for pairs.

```
fstI :: CR (a, b) (Maybe a)
fstI = MkCR (\ (a, b) -> (Just a, \ (Just a') -> (a', b)))

sndI ::  CR (a, b) (Maybe b)
sndI = MkCR (\ (a, b) -> (Just b, \ (Just b') -> (a, b')))
```

Here `Just` is used in order to emphasize that `Just a` is (a pointer to) a cell containing only one field which, in turn, represents `a`. In other words, we need to perform a dereference to obtain `a` from `Just a`.

```
data Maybe a = Just a | Nothing
```

The constructor `Nothing` may be useful for erroneous situations but is not used in the following. It is used here just to convince ourselves that `Just` is not implemented as noop.

In a naïve implementation, we allocate a new `Just` cell, pass it to a state transformer and then copy the content of the modified `Just` cell into the corresponding field. However, since state transformers are designed so that they update states destructively and the modified `Just` cell is not used elsewhere, we can implement `fstI` and `sndI` as functions which takes as an argument a pointer to a pair and then simply add the corresponding offsets then pass the result to state transformers. Generally, composable references which represent the location of a field can be implemented as functions from a pointer to a pointer.

What kind of composable references can be offered depends on the layout of data structures. We make some assumption about the representation of data structures. A constructor with n arguments is represented as a contiguous memory area of length n plus those for the header. Components are laid out in the same order as the user declared. Tags are included in the header area. When tags are not necessary to distinguish variants, tags are not used. For example, *List* ($\square$) has two variants *Cons* (`:`) and *Null* ($\square$). But *Null* has no argument and therefore does not need to be heap-allocated. It can be represented as a small integer (typically, 0) that can be distinguished from pointers to heap-allocated cells (*Cons* cells, in this case).

2.3 Composable references as cast operators

It is also possible to regard the first and the second components of a triple as a pair and define the corresponding composable reference as:

```
pair :: CR (a, b, c) (a, b)
pair = MkCR (\ (a, b, c) -> ((a, b), \ (a', b') -> (a', b', c)))
```

Then this can be also implemented without copying and be used to cast a triple into a pair; it allows a substructure of a (mutable) triple to be passed to a state transformer which expects a pair as a state and therefore acts as a cast operator. Such an ability is essential for mutable data structures. Otherwise substructures of mutable objects must be explicitly copied, which reduces the usefulness of mutable data structures significantly.

In practice, we need to name a primitive composable reference for each field when we define a new data structure. For example, a possible syntax would be as follows.

```
record Point  = MkPoint (xCoord :: Int) (yCoord :: Int)
record ColoredPoint = MkCPoint {inherit pointOf :: Point} (color :: Color)
```

```
class s :: PairClass a b where l_1 :: CR s a l_2 :: CR s b

instance (a, b) :: PairClass (Maybe a) (Maybe b) where
   l_1 = MkCR (\ (a, b) -> (Just a, \ (Just a') -> (a', b)))
   l_2 = MkCR (\ (a, b) -> (Just b, \ (Just b') -> (a, b')))

class s :: PairClass a b => s :: TripleClass a b c where
   l_3 :: CR s c

instance (a, b, c) :: PairClass (Maybe a) (Maybe b) where
   l_1 = MkCR (\ (a, b, c) -> (Just a, \ (Just a') -> (a', b, c)))
   l_2 = MkCR (\ (a, b, c) -> (Just b, \ (Just b') -> (a, b', c)))

instance (a, b, c) :: TripleClass (Maybe a) (Maybe b) (Maybe c) where
   l_3 = MkCR (\ (a, b, c) -> (Just c, \ (Just c') -> (a, b, c')))
```

Figure 5: Codes for tuples

Then primitive composable references `xCoord`, `yCoord`, `pointOf` and `color` should be generated automatically. For example,

```
pointOf :: CR ColoredPoint Point
```

We intend here that `ColoredPoint` is not isomorphic to `(Point, Color)` but to `(Int, Int, Color)`.

In many cases, we can avoid the typing problems of heterogeneous aggregates as far as their elements can be treated as references because we can cast them into a single data type to hide unnecessary parts and because we do not need to extract elements from such aggregates after their contents are modified. Instead, we can access the modified state via the same references which are initially passed to aggregates.

Suppose we have `pt :: CR s Point` and `cpt :: CR s ColoredPoint`, then `pts = [pt, cpt-*>pointOf]` has type `[CR s Point]`.

It seems desirable to express `pts` without explicit coercion as `pts = [pt, cpt]`. But this will need an extension of the type system.

In [Läu93], Läufer used first-class abstract types in order to express heterogeneous aggregates. However, we can not access additional fields any longer once they are packed as an abstract type. We avoid this problem by regarding elements of aggregates as references and using composable references to cast elements.

2.4 Parametric type classes

We can use parametric type classes as is shown in Figure 5 in order to overload composable references which are used to access fields. The notion of parametric type classes [CHO92] is a generalization of type classes. We write a *placeholder* type variable (a type variable which is constrained by the class) before "`::`" and *parameters* after the name of the class.

We will use hereafter `l_1`, `l_2`, `l_3` and so on for references of components of n-tuples, as in SML we can use `#1`,`#2` as field selectors for n-tuples.

In the original proposal, there can be no more than one instance declaration of the same top level type constructor for the same type class even if their parameters differ. For example, the following is not allowed:

```
instance [Int]  :: List Int  where ...
instance [Char] :: List Char where ...
```

This is a restriction specific to Haskell-style type classes and can be eased when used in Gofer-style type classes [Jon92]. Therefore we will not follow this restriction in the following. We can declare types with the same top level constructor as instances of a class, as far as they do not overlap. Otherwise, we would need superfluous type constructors.

2.5 Read and write operators

Primitive state transformers which read and write states are specified as follows. We will use these operators and composable references to define state transformers for general mutable data structures.

```
fetch :: ST (Maybe a) a
fetch    = MkST (\ (Just a) -> (a, Just a))

assign :: a -> ST (Maybe a) ()
assign a = MkST (\ (Just _) -> ((), Just a))

read :: SR (Maybe a) a
read     = MkSR (\ (Just a) -> a)
```

In practice, these operators are offered as primitives so that they can do in-place updating. We must ensure that `fetch` should complete dereference before the state is modified by the succeeding state transformers. A similar attention should be paid to `read` when it is used within the following operator.

```
-- a monad morphism from SR to ST
ro :: SR s a -> ST s a
ro (MkSR r) = MkST (\ s -> (r s, s))
```

We will need *freeze* operators which make a snapshot of a mutable object for each data type. The precise implementations differ according to the type; they depend on the size of objects. Therefore we should declare a type class. Note that `freeze` is an overloaded version of `freezeArr` of [LP94].

```
class Plastic s where
   freeze  :: ST s s
   replace :: s -> ST s ()
   freezeR :: SR s s
```

Then we declare instances for this class.

```
instance Plastic (a, b) where
   freeze = l_1 -!> fetch `thenST` \ a ->
            l_2 -!> fetch `thenST` \ b ->
            returnST (a, b)
```

```
replace (a, b) = l_1 -!> assign a `thenST` \ _ ->
                 l_2 -!> assign b
freezeR = ...
```

We need primitive state readers that inspect tags when we define this kind of operators for variant (union) types. However, we will not consider general variant types in what follows. We will only consider data types which have only two constructors one of which represents the "Null" value. We assume that the following overloaded operator is defined for such data types.

```
class Null s where
   nullSR :: SR s Bool
```

If the state is null, `nullSR` returns `True`. Note that we offer no operator which can change tags of mutable variant data structures so that we can always pass state transformers pointers which directly point to a contiguous memory area consisting of fields, not to a cell containing a pointer to such an area. Then the representation of "pointers" which are passed to state transformers is the same as ordinary (non-mutable) data structures and pointers to structures (records) in imperative languages such as C.

Practically instance declarations of `Plastic` should be generated automatically by the compiler (as those of the `Eq` class).

2.6 Create operator

What about the operator which creates new mutable objects? We will consider the type of the global state (state passed by `runST`) as a special type which can *contain*, as substructures, any type of mutable data structures. We offer the `melt` operator as a generalization of `newVar`.

```
class Plastic a => Contains s a where
   melt :: a -> ST s (CRV s a)
```

We should also think of `melt` as overloaded since it makes a fresh copy of its argument and returns a pointer to the copy. Of course, if the argument of `melt` is a constructor application, it would be possible to avoid copying.

`Contains s a` means that the type `s` is able to create and contain new mutable objects of type `a` as substructures. We assume that the type of the global state has this relation to any type. The typing rule for `runST` should also take this into account; the type of the state of the state transformer passed to `runST` can be constrained by relations of the form `Contains _ a`. Note that it can not be an instance of `Plastic`. Otherwise the whole global state could be copied.

Global references (references created by the `melt` operator) can be expressed as simple pointers because they do not need to be relative to something. Therefore we can define equality as pointers among them. We provide the type `CRV` in order to represent simple pointers in the global state. More precisely, `CRV s t` is a primitive data type with the following operators:

```
eqCRV   :: CRV s t -> CRV s t -> Bool
unitCR  :: CRV s t -> CR s t
```

On the other hand, `CR` is implemented as functions and may involve computations. For example, the following operator might be useful.

```
deref :: SR s (CR s a) -> CR s a
deref (MkSR sr) = MkCR (\ s -> let (MkCR r) = sr s in r s)
```

Then we can think of MutVar type of [LP94] as a CRV to a record with only one component.

```
type MutVar s a = CRV s (Maybe a)

newVar a = melt (Just a)
writeVar v a = unitCR v -!> assign a
readVar  v   = unitCR v -!> fetch
```

2.7 Mutable lists

We define the type of mutable lists as follows.

```
record ListP s a = ConsP (headP :: a) (tailP :: CRV s  (ListP s a)) | NullP
```

The tail part must be a reference because we would like the cons cell which is pointed by the tail part to be also mutable. Then two or more mutable lists can share the same mutable list as their parts, and mutable lists can be even cyclic.

As is discussed before, CRV is a direct pointer to another cons cell. Therefore the type above has the representation almost identical to list structures used in C, that is, the desirable representation in Figure 2.

For example, let us consider the following program.

```
melt NullP        'thenST' \ n0 ->
melt (ConsP 1 n0) 'thenST' \ c1 ->
melt (ConsP 2 c1) 'thenST' \ c2 ->
melt (ConsP 3 c1) 'thenST' \ c3 ->
...
```

In this example, c1 is referred to by both c2 and c3. If the ConsP cell which is pointed by c1 is modified, this change can be observed from c2 and c3. We can make them even cyclic.

```
unitCR c1-*>tailP-!>assign c2
```

Representing mutable lists in this way is, of course, low-level. However this seems necessary when we would like to handle mutable graph structures efficiently.

3 Virtual record

There is a subtle problem in handling ListP. A ConsP cell refers to another ConsP cell via a global reference. To access the tail part of a ConsP cell, we must access the global state. Let us, for example, define the "foreach" function for CR s (ListP s a).

```
foreachP :: ST (Maybe a) b -> CR s (ListP s a) -> ST s [b]
foreachP st p = ro (p -?> nullSR) 'thenST' \ b  ->
                if b then returnST [] else
                   (p-*>headP-!>st          'thenST' \ a  ->
                    p-*>tailP-!>fetch       'thenST' \ pv ->
                    foreachP st (unitCR pv) 'thenST' \ as ->
                    returnST (a:as))
```

Here, we must pass around composable references and must access the global state since a `ConsP` cell can behave as a list only when it is used with the global state. This clutters up the program. We would like to hide such a detail and like to make it explicit that we are manipulating a mutable list.

We may notice that `\ p -> p-*>tailP-!>fetch` plays a role similar to composable references of type `CR (ListP s a) (ListP s a)` for it can make another composable reference of type `CRV s (ListP s a)` from that of `CRV s (ListP s a)`. It seems necessary to unify such operators with ordinary composable references. One solution is to always regard composable references as functions from a global reference to another global reference and to pass such global references using a combination of the monad of state readers and the monad of state transformers to hide the details. However, it would be better to treat them as if both were composable references and use the operators we introduced so far. Then it is as if we were treating a mutable list which contains another mutable list as its substructure. We will pursue this strategy in what follows.

A similar problem arises when we deal with objects of type `[CR s a]`. In this case, lists themselves are not mutable, but they contain references to mutable objects as elements. We can also define "foreach" for them.

```
foreachV :: ST a b -> [CR s a] -> ST s [b]
foreachV st []     = returnST []
foreachV st (p:ps) = p-!>st          `thenST` \ a  ->
                     foreachV st ps `thenST` \ as ->
                     returnST (a:as)
```

Here lists of composable references are also passed explicitly. Another question is whether we can *cast* `CRV s (ListP s a)` and `[CR s b]` into a single data type and use them uniformly.

To achieve these goals, we use mutable data structures which are accessed indirectly. We call such indirectly accessed records *virtual records*. A virtual record is a record with an explicit method table. Then we access the state indirectly via the table. For this purpose, we define the following data type:

```
data Virtual s v = MkVirt v s
```

We define composable references for `Virtual` by using indirect access via a table (represented as `v` above). We use `unVirtCR` to extract the object which is indirectly pointed, `replTbl` to make another virtual record by replacing the indirect table by another one which is computed both from the table part and the state part. Note that method tables of virtual records are never modified by state transformers.

We will not declare virtual records as instances of `Plastic` because `fetch` and `assign` for them would be meaningless.

We define the following type class:

```
class t :: Null => t :: ListClass a where
   first :: CR t a
   rest  :: CR t t

foreach :: (s :: ListClass a) => ST a b -> ST s [b]
foreach st = ro nullSR `thenST` \ r ->
             if r then returnST []
             else first-!>st        `thenST` \ x ->
```

```
virtual   :: v -> CR s (Virtual s v)
virtual r = MkCR (\ s -> (MkVirt r s, \ (MkVirt _ s') -> s'))

unVirtCR :: (a -> CR s b) -> CR (Virtual s a) b
unVirtCR f = MkCR (\ (MkVirt a s) ->
              let MkCR r = f a; (b, b2s) = r s
              in (b, \ b' -> MkVirt a (b2s b')))

unVirtSR  :: (a -> SR s b) -> SR (Virtual s a) b
unVirtSR f = MkSR (\ (MkVirt a s) -> let MkSR sr = f a in sr s)

replTbl :: (a -> SR s b) -> CR (Virtual s a) (Virtual s b)
replTbl f = MkCR (\ (MkVirt a s) ->
            let MkSR sr = f a
            in (MkVirt (sr s) s, \ (MkVirt _ s') -> MkVirt a s'))
```

Figure 6: Operators for `Virtual` data type

```
                rest-!>foreach st `thenST` \ xs ->
                returnST (x:xs)
```

We can not declare `ListP` itself as an instance of `ListClass`. However, we can use a virtual record of `ListP` and the global state, and then declare it as an instance of `ListClass`.

```
type ListPV s a = Virtual s (CR s (ListP s a))

instance ListPV s a :: Null where
   nullSR = unVirtSR (\ p -> p-?>nullSR)

instance ListPV s a :: ListClass (Maybe a) where
   first = unVirtCR id -*> headP
   rest  = replTbl (\ p -> p-*>tailP-?>read `thenSR` \ n ->
                           returnSR (unitCR n))
```

Here `replTbl` is used in order to convert `\ p -> p-*>tailP-?>read` into a composable reference for virtual records. We must use virtual records in many cases since most objects have references to other objects as `ListP`. However simple composable references introduced in the previous section will be useful for optimization by making it explicit that some part of a stateful program can affect only limited part of the state.

We can also declare a virtual record of a simple (immutable) list of references of the form `[CR s a]` as an instance of `ListClass`.

```
type ListV s a = Virtual s [CR s a]

asListV :: CR (ListV s a) (ListV s a)
asListV = idR
```

```
instance ListV s a :: Null where
   nullSR =  unVirtSR (\ xs -> returnSR (null xs))

instance ListV s a :: ListClass a where
   first = unVirtCR (\ (h:t) -> h)
   rest  = replTbl (\ (h:t) -> returnSR t)
```

Suppose we have `pt :: CR s Point` and `cpt :: CR s ColoredPoint`, we can use the following as a reference to a list of `Point`.

```
pts = virtual [pt, cpt-*>pointOf] -*> asListV
```

In this expression, `-*> asListV` is necessary because the type inference algorithm infers too general a type and fails to conclude that it is an instance of `ListClass`.

We can freely interleave `pts` with `pt` and `cpt`. For example:

```
pts -!> foreach (move 1 2)    'thenST' \ _ ->
pt  -*> xCoord -!> fetch      'thenST' \ a ->
cpt -*> color -!> assign Red 'thenST' \ b ->
pts -!> foreach (move 3 1)
```

where `move :: Int -> Int -> ST Point ()`. The effect of actions on `pts` are visible from `pt` and `cpt`, and vice versa.

As an attempt to standardize two representations of `ListClass`, for example, we define the following data type.

```
data ListS s a = MkListS (SR s Bool) (CR s a) (SR s (ListS s a))
type ListSV s a = Virtual s (ListS s a)
```

We can declare `ListSV s a` as an instance of `ListClass` and coerce both `[CR s ...]` and `CR s (ListP s ...)` into this data type. This is shown in Figure 7. Then it is possible to handle heterogeneous lists of lists, for example, it is possible to apply `foreach (foreach (move 1 0))` to something like `virtual [coerceP ..., coerceV ...]`.

As is shown above, virtual records seem to be useful for standardization of mutable data structures. For example, we define for `PairClass`:

```
type PairV s a b = Virtual s (CR s a, CR s b)

asPairV :: CR (PairV s a b) (PairV s a b)
asPairV = idR

instance PairV s a b :: PairClass a b where
   1_1 = unVirtCR (\ (a, b) -> a)
   ...

pairV :: (x :: PairClass a b) => CR s x -> CR s (PairV s a b)
pairV xr = virtual (xr-*>1_1, xr-*>1_2) -*> asPairV
```

```
instance ListSV s a :: Null where
   nullSR = unVirtSR (\ (MkListS n _ _) -> n)

instance ListSV s a :: ListClass a where
   first = unVirtCR (\ (MkListS _ f _) -> f)
   rest  = replTbl (\ (MkListS _ _ r) -> r)

coerceP :: CRV s (ListP s a) -> ListS s (Maybe a)
coerceP pv = let p = unitCR pv
             in MkListS (p-?>nullSR) (p-*>headP)
                        (p-*>tailP-?>read `thenSR` \ pv' ->
                         returnSR (coerceP pv'))

coerceV :: [CR s a] -> ListS s a
coerceV xs = MkListS (returnSR (null xs)) (head xs)
                     (returnSR (coerceV (tail xs)))
```

Figure 7: Instance declarations and operators for **ListSV**

Then we can cast all the instances of **PairClass** into **PairV** using **pairV**. This may be necessary because we can not cast all the instances of **PairClass** into **Pair** because the two fields need not be laid out contiguously.

It would be desirable if we could define a type (constructor) of standard method vectors v_R, whenever we declare a parametric type class R which is intended as a "record" class. Then we could declare *Virtual* s $(v_R\ s)$ as an instance of R. We could use this form as the standard representation of a parametric type class. However, it is not clear how we should define such method vectors for general classes especially for classes which express recursive mutable data structures.

4 Conclusion

We proposed composable references in order to deal with data structures with mutable components. Mutable data structures can be represented as efficiently as those used in conventional languages. Especially, mutable lists can be represented without indirect nodes. As a consequence, we can enjoy both the flexibility of references and the conciseness of records.

Composable references can be also used to cast mutable objects. This is useful when otherwise heterogeneous aggregates are necessary.

We introduced virtual records in order to handle data structures which contain references as their fields. Virtual records are used to hide details and to treat them in a higher level.

Garbage collections might be problematic because we use pointers into the middle of data structures. This might be avoided by passing a pointer to the header and an index from the header separately.

Acknowledgement

The author would like to thank Yasuhiko Minamide and Atsushi Ohori for their helpful comments. He is also grateful to the anonymous SIPL'95 reviewers for their detailed comments.

References

[AvGP93] Peter Achten, John van Groningen, and Rinus Plasmeijer. High level specification of I/O in functional languages. In Launchburyet al., editors, *Proceedings of Glasgow Workshop on Functional Programming*. Springer Verlag, 1993.

[CHO92] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *ACM Conf. on LISP and Functional Programming*, June 1992.

[Jon92] Mark P. Jones. A theory of qualified types. In *ESOP '92: European Symposium on Programming, Rennes, France*, New York, February 1992. Springer-Verlag. Lecture Notes in Computer Science, 582.

[Läu93] Konstantin Läufer. An extension of Haskell with first-class abstract types, September 1993. Submitted to *the Journal of Functional Programming.*

[Ler93] Xavier Leroy. *The Caml Light system, release 0.6 Documentation and user's manual.* INRIA, September 1993.

[LP94] John Launchbury and Simon L. Peyton Jones. Lazy functional state threads. In *Programming Languages Design and Implementation (PLDI) '94*, June 1994.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *IEEE Symposium on Logic in Computer Science*, June 1989.

[PJW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Annual ACM Symp. on Principles of Prog. Languages*, 1993.

[Wad90] Philip Wadler. Comprehending monads. In *ACM Symp. on Lisp and Functional Programming*, pp. 61–78, 1990.

[Wad92] Philip Wadler. The essence of functional programming. In *Annual ACM Symp. on Principles of Prog. Languages*, 1992.

Applying π: Towards a Basis for Concurrent Imperative Programming

Martin Odersky

Universität Karlsruhe
76128 Karlsruhe, Germany
odersky@ira.uka.de

December 22, 1994

Abstract

We study an extension of asynchronous π-calculus where names can be returned from processes. We show that with this simple extension an extensive range of functional, state-based and control-based programming constructs can be expressed by macro expansions, similar to Church-encodings in lambda calculus.

1 Introduction

Most programming languages in use today have some way to express concurrent execution of processes – either in the language itself (e.g. Ada [20], Modula-3 [5], Facile [7], CML [23]) or by means of a library (e.g. Modula-2's Process module [28], C++'s thread library [25]). This paper proposes a formal basis for reasoning about such languages.

Traditionally, formal foundations for languages with concurrency constructs come in one of two styles. Most commonly, one combines a semantic description for the sequential base language with another one for the concurrency primitives. For instance, semantic descriptions of Facile [7] or CML [2] define a structured operational semantics for the base language as a special case of a larger labeled transition system that also models the concurrent aspects of the language. This style of description has the advantage that a semantics of the sequential part of the language can be obtained by subsetting. However, the resulting formal systems tend to be large.

Alternatively, one can use a standard process calculus such as CCS [13] or π-calculus [16] to reason about both the base language and the concurrency primitives. An example of this approach is the PICT programming language [21] that was designed with asynchronous π-calculus [4] as a basis. PICT stays fairly close to the underlying calculus and consequently does not fully support sequential programming constructs such as functions or sequential composition. Representing these constructs in traditional process calculi requires a *global* encoding, not unlike a conversion to continuation passing style in functional programming[1]. Examples of such encodings are found in work by Milner

[1] In fact, PICT takes an intermediate approach: There is a notation for function definition, which leaves the result channel implicit, but there is no corresponding notation for function calls, so that an explicit result channel argument always needs to be passed. In effect, this leads to function definitions in PICT being CPS-converted one at a time.

[15] for the case of functions and by Walker [27] or Jones [11] for the case of objects. If our aim is to reason about source programs such encodings are undesirable since they are all-or-nothing propositions: To reason about one part of a program one must encode everything.

We would prefer a relationship between programming language and foundation that is similar to the relationship between functional languages and λ-calculus. There, one is transformed to the other via *Church-encodings*, which are pure macro expansions. In this paper we show that a modest change to a standard process calculus is sufficient to capture both call-by-value functional programming and imperative programming via similar encodings. The *applied π calculus* augments asynchronous π calculus [4] (which is essentially equivalent to ν-calculus [10]) with the ability to return a name from a process. Together with standard name restriction this gives us a way to model anonymous values in the calculus. It turns out that this is all that is needed to encode essentially all sequential programming constructs in a concise and straightforward manner.

Interestingly, with just seven term formation rules and one reduction rule, applied π is more compact than calculi for sequential state-based languages [12, 6, 19]. This comparison is not completely fair, however, since the encoding into applied π gives us only an operational understanding of functional and imperative constructs. Much less is known at present about the observational properties of the encodings. In general, process contexts discriminate more terms than sequential contexts. Hence, source language constructs would need to be encapsulated in some way in order to preserve their observational properties, but such an encapsulation is not discussed here. Nevertheless, we believe that applied π can be useful for gaining semantic intuition about how familiar functional and state-based programming constructs should behave when extended to a concurrent setting.

Related work. We have already mentioned the work on PICT and the encodings by Milner, Walker, and Jones. Sangiorgi has argued that the higher-order π calculus improves on first-order π calculus as a foundation for functional programming [24]. In a sense, applied π's ability to return a name from a process is an alternative to higher-order processes, since λ-abstractions can be represented. Boudol's γ-calculus [3] tries to generalize both CCS and λ-calculus. Like in λ — and unlike in applied π — communicating agents are matched by position rather than just channel name.

Our process equivalence relation is based on Milner's and Sangiorgi's barbed bisimulation [17]. We adapt their definitions in a straightforward way to the asynchronous and applied case. Honda and Yoshida [9] have shown for an asynchronous calculus that barbed bisimulation has a tractable characterization that does not depend on a quantification over contexts.

The rest of this paper is organized as follows. Section 2 presents an operational semantics for applied π. Section 3 defines a notion of process equivalence for the calculus. Section 4 shows how functional programming constructs can be encoded in applied π. Section 5 does the same for imperative programming, giving encodings for the essential constructions of state and control. Section 6 presents an encoding of applied π in asynchronous π. Section 7 concludes.

2 The Core Calculus

Syntactic Domains

Variables	x, y, z			
Preterms	M, N, P	$=$	x	Variable
		$\mid$	$\nu x.M$	Restriction
		$\mid$	$x^?y.M$	Abstraction (Input)
		$\mid$	xM	Application (Output)
		$\mid$	$M \mid N$	Parallel Composition
		$\mid$	$!M$	Replication
		$\mid$	0	Identity

We build on asynchronous π-calculus [4], modulo some minor notational modifications that are introduced for making the treatment of function application smoother. There is one extension: Processes may evaluate to names, and an arbitrary term instead of a single name may appear as the argument of an application. Roughly speaking, an application xM is evaluated by evaluating M to some number of names which are all passed in parallel to the channel x.

Hence, applied π is a rather small variation of a standard process calculus. However, it can also be seen as a generalization of λ-calculus where the concept of a λ-abstraction is generalized in two ways. First, applied π's abstractions can be used only once, unless they are prefixed by a (!) replicator. This is similar to the role of abstractions in linear λ-calculus [1]. Second, an abstraction and its argument are matched by name rather than by position. Fresh local names are introduced by a restriction prefix νx (this has also been studied in the context of λ-calculus [22, 18]).

Notational Conventions. $\mathrm{fn}(M)$, the set of free names of a term M, is given by

$$\begin{array}{lcllcl}
\mathrm{fn}(x) & = & \{x\} & \mathrm{fn}(\nu x.M) & = & \mathrm{fn}(M)\backslash\{x\} \\
\mathrm{fn}(x^?y.M) & = & \{x\} \cup (\mathrm{fn}(M)\backslash\{y\}) & \mathrm{fn}(xM) & = & \{x\} \cup \mathrm{fn}(M) \\
\mathrm{fn}(M \mid N) & = & \mathrm{fn}(M) \cup \mathrm{fn}(N) & \mathrm{fn}(!M) & = & \mathrm{fn}(M) \\
\mathrm{fn}(0) & = & \{\}. & & &
\end{array}$$

$[x/y]M$ denotes substitution of x for all free occurrences of y in M. To avoid name capture problems in substitutions, we assume everywhere that the free and bound variables of a term and all its subterms are distinct. This can always be achieved by α-renaming (see below).

A note on precedence: Application binds tightest, followed by replication (!) and the binding prefixes νx and $x^?y$, followed by parallel composition ($\mid$). Application is left-associative and parallel composition is associative. Grouping can be changed by using parentheses.

We also use the words *channel* and *agent* interchangeably for name and term, respectively. We sometimes contract multiple input or restriction prefixes, using the abbreviations

$$\begin{array}{lcl}
\nu x_1 \ldots x_n.M & \stackrel{\mathrm{def}}{=} & \nu x_1. \ldots \nu x_n.M \\
x^?y_1 \ldots y_n.M & \stackrel{\mathrm{def}}{=} & x^?y_1. \ldots x^?y_n.M \ .
\end{array}$$

Equivalences

Terms are equivalence classes of preterms. We take *syntactic equivalence* ($\equiv$) to be the smallest congruence that satisfies the laws below.

1. Variables can be α-renamed.

(α_1)	$\nu x.M$	$\equiv$	$\nu y.[y/x]M$	$(y \notin \mathrm{fn}(M))$
(α_2)	$z^?x.M$	$\equiv$	$z^?y.[y/x]M$	$(y \notin \mathrm{fn}(M))$

2. Replication composes arbitrarily many copies of a term in parallel.

(Repl)	$!M$	$\equiv$	$M \mid !M$

3. Parallel composition is commutative and associative, with identity 0.

(Comm)	$M \mid N$	$\equiv$	$N \mid M$
(Assoc)	$(M \mid N) \mid P$	$\equiv$	$M \mid (N \mid P)$
(Id)	$M \mid 0$	$\equiv$	M

4. The scope of a restricted variable x can be extended over parallel composition and application, provided x is not captured. Restriction with an unused name has no effect.

(ν-Par)	$M \mid \nu x.N$	$\equiv$	$\nu x.(M \mid N)$	$(x \notin \mathrm{fn}(M))$
(ν-Apply)	$y(\nu x.M)$	$\equiv$	$\nu x.yM$	$(x \neq y)$
(ν-Garbage)	$\nu x.M$	$\equiv$	M	$(x \notin \mathrm{fn}(M))$

5. Application distributes over parallel composition. Application has no effect on abstraction arguments.

(Dist)	$x(M \mid N)$	$\equiv$	$xM \mid xN$
(Absorb)	$x(y^?z.M)$	$\equiv$	$y^?z.M$

Except for (ν-Apply), equalities (1)–(4) all have counterparts in π-calculus. Equalities (Dist) and (Absorb) are perhaps surprising at first. Essentially, they introduce a fundamental asymmetry between applications and abstractions. Abstractions are *volatile*, in that they can move freely into and out-of applications. By contrast, applications are *stationary*, they appear only a single fixed context. Note the similarity to first order functional programming, where abstractions correspond to function definitions (where the location of the definition does not matter) and applications correspond to function calls (where the point of call does matter). However, unlike in functional programming, an abstraction can be used only once if it is not replicated. We will see that this resource-consciousness is the essential ingredient that allows applied π to model side-effects in expressions.

Reduction

There is a single reduction rule.

(Reaction)	$x^?y.M \mid xz$	$\rightarrow$	$[z/y]M$

Reduction is considered modulo syntactic equivalence. Reduction can be applied anywhere in a term except under an abstraction or a replication. That is, a binary reduction relation ($\rightarrow$) between terms is given by the axiom (Reaction) and the inference rule

$$\text{(Context)} \qquad \frac{M' \equiv M \qquad M \rightarrow N \qquad N \equiv N'}{E[M'] \rightarrow E[N']}$$

where E is an arbitrary *evaluation context* that can be generated by the grammar

$$E \;=\; [\,] \;\mid\; \nu x.E \;\mid\; xE \;\mid\; E|M \;.$$

Let $\twoheadrightarrow$ be the reflexive transitive closure of reduction.

3 A Process Equivalence

Our notion of equivalence of applied π terms is based on bisimulation. The central intuition of bisimulation is that an experiment which tests whether two processes are equivalent can be constructed from two basic actions: One can observe the interaction of a running process, and one can freeze a process in a given state and let in run repeatedly starting from this state. The latter distinguishes bisimulation from trace equivalence.

For processes whose operational semantics is defined by means of a reduction relation, a particularly simple form of bisimulation can be devised, which tests only the possibility of interacting on a channel, but disregards what is communicated over it. This relation is called *barbed bisimulation* [17]. For applied π, barbed bisimulation can be simplified further in that only the action of returning a name, but not input or output actions, can be observed. This is formalized in the following definitions.

Definition. A symmetric relation $\mathcal{R}$ on terms is *reduction-closed* iff $M\mathcal{R}N$ and $M \twoheadrightarrow M'$ implies the existence of a term N' such that $N \twoheadrightarrow N'$ and $M'\mathcal{R}N'$.

Definition. A term M *converges*, written $M \Downarrow$, if $M \twoheadrightarrow (x \mid N)$, or $M \twoheadrightarrow \nu x.(x \mid N)$ for some name x and term N.

Definition. A symmetric relation $\mathcal{R}$ between terms is a *(weak) barbed bisimulation* for applied π iff $\mathcal{R}$ is reduction-closed and $M\mathcal{R}N$ and $M \Downarrow$ implies $N \Downarrow$. $M \dot{\approx} N$ iff there is a bisimulation $\mathcal{R}$ such that $M\mathcal{R}N$.

$\dot{\approx}$ is not a congruence, for instance it is not preserved by parallel composition: $x^?y.0 \dot{\approx} x^?y.y$, but not $x^?y.0 \mid xz \dot{\approx} x^?y.y \mid xz$. We therefore define:

Definition. Let $\approx$ be the largest congruence such that $\approx \subseteq \dot{\approx}$.

In the following, whenever we say that two terms M, N are equivalent (written $M = N$) we mean that they are barbed congruent, *i.e.* $M \approx N$.

Proposition 3.1 (*a*) The following are bisimulation equivalences in applied π.

$$x(!M) = !xM \quad (1)$$
$$!M = !M \mid !M \quad (2)$$
$$\nu x.x^?y.M = 0 \quad (3)$$
$$\nu x.(xy \mid x^?z.M) = \nu x.[y/z]M \quad (4)$$
$$\nu x.(xy \mid !x^?z.M) = \nu x.[y/z]M \quad (5)$$

(*b*) If $x \notin \mathrm{fn}(M, N, P)$ then the following are also bisimulation equivalences.

$$\nu x.(xM \mid x^?y.zy) = zM \quad (6)$$
$$\nu x.(xM \mid xN \mid !x^?y.P) = \nu x.(xM \mid !x^?y.P) \mid \nu x.(xN \mid !x^?y.P) \quad (7)$$
$$\nu x.(!xM \mid !x^?y.P) = !\nu x.(xM \mid !x^?y.P) \quad (8)$$

Equation (1) is the analogue of the (Absorb) equivalence for replicated abstraction. Equation (2) says that parallel composition is idempotent on replicated terms. Equation (3) says that any term that reads from a freshly allocated variable is an identity for parallel composition. We also call such terms *inert*. Equations (4) and (5) say that reduction via a local variable is an equivalence. Equation (6) says that forwarding a term via a local variable is equivalent to sending the term directly to its final destination. Finally, equations (7) and (8) are factoring laws for a parallel composition or replication of output terms in a local computation.

4 Encoding Functions

We now encode functional programming constructs in applied π, using just macro expansions. We define an affine λ-abstraction $\lambda_1 x.M$, which can be applied at most once, and an unrestricted call-by-value abstraction $\lambda x.M$.

$$\lambda_1 x.M \stackrel{\text{def}}{=} \nu f.(f \mid f^?x.M) \qquad (f \text{ fresh})$$
$$\lambda x.M \stackrel{\text{def}}{=} \nu f.(f \mid !f^?x.M) \qquad (f \text{ fresh})$$

General function application can be simulated by using a local name for the function part of the application. Here we have a choice, whether function and argument part should be evaluated concurrently or in sequence. We start with sequential application, which is expressed by juxtaposition of function and argument and is encoded as follows:

$$M\,N \stackrel{\text{def}}{=} \nu x.(xM \mid !x^?f.fN) \qquad (x, f \text{ fresh})$$

Application of a channel x is (modulo $=$) a special case of sequential function application, as is seen by looking at the expanded form of xM, *i.e.* $\nu y.(yx \mid !y^?f.fM)$, where y and f are fresh names.

$$\begin{aligned} &\nu y.(yx \mid y^?f.fM) \\ &= \nu y.xM && \text{by (5)} \\ &\equiv xM && \text{by } (\nu\text{-GC}) \end{aligned}$$

This explains why we have chosen to use $x^?$ for abstraction and plain x for application, whereas in original π calculus plain x is an input prefix amd $\overline{x}$ is an output prefix.

Example 4.1

$$
\begin{array}{lll}
(\lambda_1 x.x)(\lambda_1 y.y) & & \\
\quad \stackrel{\text{def}}{=} & \nu a.(a(\nu g.(g \mid g?x.x)) \mid a?g.gH) & \text{where } H \stackrel{\text{def}}{=} \nu h.(h \mid h?y.y) \\
\quad \equiv & \nu a.\nu g.(a(g \mid g?x.x) \mid a?g.gH) & \text{by } (\nu\text{-}*) \\
\quad \equiv & \nu a.\nu g.((ag \mid g?x.x) \mid a?g.gH) & \text{by (Dist), (Absorb)} \\
\quad \equiv & \nu a.\nu g.((ag \mid a?g.gH) \mid g?x.x) & \text{by (Assoc), (Comm)} \\
\quad \rightarrow & \nu a.\nu g.(gH \mid g?x.x) & \text{by reduction} \\
\quad \equiv & \nu a.\nu g.(g(\nu h.(h \mid h?y.y)) \mid g?x.x) & \text{substituting the definition of } H \\
\quad \equiv & \nu a.\nu g.\nu h.(g(h \mid h?y.y) \mid g?x.x) & \text{by various } (\nu) \text{ equivalences} \\
\quad \equiv & \nu a.\nu g.\nu h.((gh \mid h?y.y) \mid g?x.x) & \text{by (Dist), (Absorb)} \\
\quad \equiv & \nu a.\nu g.\nu h.(gh \mid g?x.x \mid h?y.y) & \text{by (Assoc), (Comm)} \\
\quad \rightarrow & \nu a.\nu g.\nu h.(h \mid h?y.y) & \text{reducing via } g \\
\quad \equiv & \nu h.(h \mid h?y.y) & \text{by } (\nu\text{-}*), \text{(GC)} \\
\quad \stackrel{\text{def}}{=} & \lambda_1 y.y & \text{by sugaring}
\end{array}
$$

Proposition 4.2 The following are observational equivalences for applied π.

$$
\begin{array}{rcll}
(M \mid N)P & = & MP \mid NP & (9) \\
M(N \mid P) & = & MN \mid MP & (10) \\
(x?y.M)N & = & x?y.M & (11) \\
(!M)N & = & !(MN) & (12)
\end{array}
$$

The proofs are all simple equivalence chains. Two examples are: (9):

$$
\begin{array}{lll}
(M \mid N)P & & \\
\quad \stackrel{\text{def}}{=} & \nu x.(x(M \mid N) \mid !x?y.yP) & \text{desugaring the application} \\
\quad \equiv & \nu x.(xM \mid xN \mid !x?y.yP) & \text{by (Dist)} \\
\quad = & \nu x.(xM \mid !x?y.yP) \mid \nu x.(xN \mid !x?y.yP) & \text{by (7)} \\
\quad \stackrel{\text{def}}{=} & MP \mid NP & \text{resugaring}
\end{array}
$$

(12):

$$
\begin{array}{lll}
(!M)N & & \\
\quad \stackrel{\text{def}}{=} & \nu x.(x(!M) \mid !x?f.fN) & \text{desugaring the application} \\
\quad = & \nu x.(!xM \mid !x?f.fN) & \text{by (1)} \\
\quad = & !\nu x.(xM \mid !x?f.fN) & \text{by (8)} \\
\quad \stackrel{\text{def}}{=} & !(MN) & \text{resugaring}
\end{array}
$$

Note that symmetric versions of (11) and (12) do not hold; *e.g.* in $M(x?y.N)$, the abstraction becomes available only after M reduces to a name.

Parallel application ($\bullet$) imposes no sequencing constraints on the evaluation of a function and its argument. It is encoded as follows.

$$M \bullet N \;=\; \nu x.\nu y.(xM \mid yN \mid x^?f.y^?a.fa)$$

Local Definitions

Using lambda abstraction and application, we can define a let-construct $\mathsf{let}\ x = M\ \mathsf{in}\ N$ to be sugar for $(\lambda x.N)\,M$. Expanding this and simplifying yields:

$$\begin{aligned}
&\mathsf{let}\ x = M\ \mathsf{in}\ N \\
&\quad \stackrel{\text{def}}{=} \quad (\lambda x.N)\,M \\
&\quad \stackrel{\text{def}}{=} \quad \nu z.(z(\nu y.(y \mid !y^?x.N)) \mid !z^?w.wM) \\
&\quad \equiv \quad \nu y.(\nu z.(zy \mid !z^?w.wM) \mid !y^?x.N) \\
&\quad = \quad \nu y.(yM \mid !y^?x.N) \qquad\qquad \text{by (5)}\ .
\end{aligned}$$

As in the λ-calculus, this gives us a non-recursive local definition where the variable x cannot appear in the body of its defining term, M. Recursive (function) definitions are also possible. They can be defined as follows:

$$\mathsf{letrec}\ f\,x = M\ \mathsf{in}\ N \;\stackrel{\text{def}}{=}\; \nu f.(N \mid !f^?x.M)\ .$$

This extends naturally to mutual recursion:

$$\begin{aligned}
&\mathsf{letrec}\ f_1\,x_1 = M_1, \ldots, f_n\,x_n = M_n\ \mathsf{in}\ N \\
&\quad \stackrel{\text{def}}{=} \quad \nu f_1 \ldots f_n.(N \mid !f_1^?x_1.M_1 \mid \ldots \mid !f_n^?x_n.M_n)\ .
\end{aligned}$$

5 Encoding Imperative Programs

Sequential Composition

We can define the sequential composition of a value-producing term M and a term N by

$$M\,;N \;\stackrel{\text{def}}{=}\; \nu x.(xM \mid x^?y.N) \qquad (x, y \text{ fresh})\ .$$

This evaluates M until a value is produced, and then continues with N. The value produced by M is discarded. We use the convention that (;) has higher precedence than (|) but lower precedence than the unary operators.

If in $(M_1 \mid \ldots \mid M_n)\,;P$ each M_i produces a value then P will be enabled as soon as one of the M_i produces its result. We can force a wait for all M_i's by defining a blocking parallel composition ($|||$) of independent subcomputations — this is essentially Hoare's interleave operator [8]. Interleave is expressed as follows:

$$M_1 ||| \ldots ||| M_n \;\stackrel{\text{def}}{=}\; \nu x_1 \ldots x_n.(x_1M_1 \mid \ldots \mid x_nM_n \mid x_1^?y_1 \ldots x_n^?y_n.())$$

Here, the empty tuple () is a shorthand that stands for some arbitrary reserved name, whose identity is unimportant.

Dereferencing

One sometimes wants to use the result of a read operation as an argument in an application. Writing $x(y?z.z)$ would not do, as this expression is equivalent to just $y?z.z$. Instead, one can use $x(y\uparrow)$ where the read operator ($\uparrow$) is given by:

$$x\uparrow \stackrel{\text{def}}{=} \nu a.(a() \mid x?y.a?z.y) \ .$$

Note the role of the *acknowledgment* channel a. Its purpose can be explained as follows. Clearly, to read from a channel x, we need a term of the form $x?y.M$. The problem is that this term is volatile, and hence will reduce in the context of the corresponding output operation. But when writing $z(x\uparrow)$, for instance, we want the read value to be passed to z. This is accomplished by the pair of the output action $a()$ in the parallel composition and the input action $a?z$ in the reader term. Figuratively an interaction via a "pulls back" the abstraction $a?z.y$ into the context of the output term $a()$. A similar technique is used below in the modeling of mutable variables.

Mutable Variables

We now encode mutable variables with an allocation operation **newref** x, where M computes the initial value of the allocated result variable, an assignment operation $r := x$, and a dereferencing operation $r\uparrow$.

$$\begin{aligned}
&\textsf{newref}\ x && \stackrel{\text{def}}{=} && \nu r.(r \mid rx) \\
&r := x && \stackrel{\text{def}}{=} && \nu a.(a() \mid r?y.(rx \mid a?z.x)) \\
&r\uparrow && \stackrel{\text{def}}{=} && \nu a.(a() \mid r?y.(ry \mid a?z.y))
\end{aligned}$$

These constructs model a mutable variable by a name r that always has a pending output operation rx, where x denotes the current value of the variable. Consequently, assignment to a mutable variable involves reading out the old value before the new value is written. Likewise, dereferencing a mutable a variable involves reading out its value and then writing it back. Note that this makes assignment and read symmetric operations, which is reflected in the similarity of their encodings.

Initializations and assignments with structured terms are derived from these encodings as in the case of functions. That is,

$$\textsf{newref}\ M \stackrel{\text{def}}{=} (\lambda x.\textsf{newref}\ x)\, M = \nu y.(y?x.\textsf{newref}\ x \mid yM) \ ,$$

and, analogously,

$$r := M \stackrel{\text{def}}{=} (\lambda x.r := x)\, M = \nu y.(y?x.r := x \mid yM) \ .$$

Multiple assignments can be expressed by interleaving.

$$r_1, \ldots, r_n := M_1, \ldots, M_n \stackrel{\text{def}}{=} r_1 := M_1 \,|||\, \ldots \,|||\, r_n := M_n \ .$$

Example 5.1 The following reduction shows that (;) enforces sequential execution of assignments. Consider the sequence of assignments $r := 1 \,;\, r := r\!\uparrow + 1$ with initial value 0 of r:

$$
\begin{array}{lll}
(r := 1 \,;\, r := r\!\uparrow + 1) \mid r0 & & \\
\quad \stackrel{\text{def}}{=} & (\nu a.(a() \mid r^?y.(r1 \mid a^?z.1)) \,;\, r := r\!\uparrow + 1) \mid r0 & \text{by desugaring the first assignment} \\
\quad \stackrel{\text{def}}{=} & \nu s.(s(\nu a.(a() \mid r^?y.(r1 \mid a^?z.1))) \mid s^?d.r := r\!\uparrow + 1) \mid r0 & \text{by expanding the sequential composition} \\
\quad \equiv & \nu s.\nu a.(s(a()) \mid r^?y.(r1 \mid a^?z.1) \mid s^?d.r := r\!\uparrow + 1 \mid r0) & \text{by various equivalences} \\
\quad \rightarrow & \nu s.\nu a.(s(a()) \mid r1 \mid a^?z.1 \mid s^?d.r := r\!\uparrow + 1) & \text{reducing via } r \\
\quad \equiv & \nu s.\nu a.(s(a() \mid a^?z.1) \mid r1 \mid s^?d.r := r\!\uparrow + 1) & \text{by (Dist), (Absorb)} \\
\quad \rightarrow & \nu s.\nu a.(s1 \mid r1 \mid s^?d.r := r\!\uparrow + 1) & \text{reducing via } a \\
\quad \rightarrow & \nu s.\nu a.(r1 \mid r := r\!\uparrow + 1) & \text{reducing via } s \\
\quad \equiv & r1 \mid r := r\!\uparrow + 1 & \text{by (GC)}
\end{array}
$$

Control

We conclude our overview of sequential programming constructs with an encoding of control operators *abort* and *call/cc* in applied π. To make a program M abortable, embed it in the context

$$\nu e.(M \mid e()) \ .$$

where e is some fresh name. Then *abort* is given by

$$abort\ x \ \stackrel{\text{def}}{=} \ e^?y.x \ .$$

Note the *reverse trigger*, $e()$, that gets replaced by the argument x of *abort* by creating the agent $e^?y.x$. Since abstractions are volatile, an occurrence of *abort* inside an application chain will thus react with the top-level trigger $e()$, thereby returning a result from the program. A similar trick is used in the encoding of *call/cc*:

$$call/cc\ f \ \stackrel{\text{def}}{=} \ \nu e.\nu k.(fk \mid !k^?x.e^?y.x \mid !e())$$

This passes a continuation k that captures the current context to the function f. Again, $e()$ acts as a reverse trigger that injects the argument of the continuation variable k into the context of the *call/cc*.

6 Encoding Applied π in Asynchronous π

Applied π has close relations to asynchronous π calculus. We now formalize this statement by giving an encoding of applied π in asynchronous π. We use a slight variation of Boudol's definition. In our version, π_{async}, terms are given by

$$M \ = \ \nu x.M \ \mid \ x^?y.M \ \mid \ xy \ \mid \ M|N \ \mid \ !M \ \mid \ 0 \ ,$$

modulo syntactic equivalences (α), (Repl), (Comm), (Assoc), (Id), (ν-Par), (ν-Garbage) and reduction is as in applied π.

As an equivalence theory for asynchronous π terms we also use barbed bisimulation, which now takes the following form.

Definition. A term $M \in \pi_{\text{async}}$ *outputs* on a channel x, written $M \Downarrow_x$, if there is a name y and a term N such that either $M \twoheadrightarrow xy \mid N$ or $M \twoheadrightarrow \nu y.(xy \mid N)$.

That is, we take as observables output actions, but not input actions. Based on this notion of observation, barbed bisimulation and barbed congruence are then defined as usual:

Definition. A symmetric relation $\mathcal{R}$ between terms in π_{async} is a (weak) asynchronous barbed bisimulation iff $\mathcal{R}$ is reduction-closed and $M\mathcal{R}N$ and $M \Downarrow_x$ implies $N \Downarrow_x$. $M \dot{\approx}_{\text{async}} N$ iff there is an asynchronous bisimulation $\mathcal{R}$ such that $M\mathcal{R}N$. let $\approx_{\text{async}}$ be the largest congruence contained in $\dot{\approx}$.

We now define a mapping $[\![\cdot]\!]$ that takes as arguments an applied π term M and a name r and yields a term in π_{async}. The name r represents a channel where the result of the translated term should be sent to. The translation is given by:

$$
\begin{array}{lcl}
[\![x]\!]r & = & rx \\
[\![\nu x.M]\!]r & = & \nu x.[\![M]\!]r \\
[\![x^?y.M]\!]r & = & x^?(y,s).[\![M]\!]s \\
[\![xM]\!]r & = & \nu s.([\![M]\!]s \mid !\nu t.s^?y.(x(y,t) \mid !t^?z.rz)) \\
[\![M \mid N]\!]r & = & [\![M]\!]r \mid [\![N]\!]r \\
[\![!M]\!]r & = & ![\![M]\!]r \ .
\end{array}
$$

We use for brevity polyadic inputs $x^?(y,z).M$ and outputs $x(y,z)$ which can be expanded with Honda and Tokoro's "zip-lock" technique[2] [10]:

$$
\begin{array}{lcl}
x^?(y,z).M & \stackrel{\text{def}}{=} & x^?u.\nu v.(uv \mid v^?y.\nu w.(uw \mid w^?z.M)) \\
x(y,z) & \stackrel{\text{def}}{=} & \nu u.(xu \mid u^?v.(vy \mid u^?w.wz)) \ .
\end{array}
$$

To show that this encoding is well-defined, have have to verify that it is insensitive to the preterm chosen to represent a term.

Proposition 6.1 Let r be a name. Let M, N be preterms such that $M \equiv N$. Then $[\![M]\!]r \approx [\![N]\!]r$.

Proof Sketch: Verify that the translations of all syntactic equivalence rules are barbed asynchronous bisimulations. □

The following lemma shows that forwarding of a result via an intermediary is indistinguishable from passing the result directly:

Lemma 6.2 Assume $s, t \notin \text{fn}(M)$. Then $[\![M]\!]r \approx \nu s.([\![M]\!]s \mid s^?x.rx)$.

[2] Note how parallel compositions in the input term correspond to input prefixes in the output term and vice versa.

We now show that the encoding preserves the reduction semantics of applied π, in the following sense:

Definition. Let $M, N \in \pi_{\text{async}}$. $M \rightarrow^{\approx}_{\text{async}} N$ iff there are terms $M' \approx_{\text{async}} M$ and $N' \approx_{\text{async}} N$ such that $M' \rightarrow_{\text{async}} N'$.

Proposition 6.3 Let M, N be terms in applied π and let $r \notin \text{fn}(M, N)$. If $M \rightarrow N$ then $[\![M]\!]r \rightarrow^{\approx}_{\text{async}} [\![N]\!]r$.

Proof: Assume $M \rightarrow N$ and $r \notin \text{fn}(M, N)$. Then we have:

$$
\begin{array}{lll}
[\![x^?z.M \mid xy]\!]r & & \\
\quad \equiv & x^?(z,s).[\![M]\!]s \mid \nu s.(sy \mid s^?z.\nu t.(x(z,t) \mid t^?u.ru)) & \\
\quad = & x^?(z,s).[\![M]\!]s \mid \nu t.(x(y,t) \mid t^?u.ru)) & \text{by local reduction} \\
\quad \rightarrow & \nu t.([y/z,t/s][\![M]\!]s \mid t^?u.ru) & \\
\quad \equiv & \nu t.([\![[y/z]M]\!]t \mid t^?u.ru) & \\
\quad = & [\![[y/z]M]\!]r & \text{by Lemma 6.2}
\end{array}
$$

□

Proposition 6.4 Let M, N be terms in applied π. If, for all $r \notin \text{fn}(M, N)$, $[\![M]\!]r \approx_{\text{async}} [\![N]\!]r$ then $M \approx N$.

Proof: Assume $M \not\approx N$. Then there is a context C such that one of $C[M]$, $C[N]$ converges but the other does not. W.l.o.g. assume that $C[M] \Downarrow$, $C[N] \not\Downarrow$. Let a be a fresh name. Then, because of Proposition 6.3, $[\![C[M]]\!]a \Downarrow_a$ but $[\![C[N]]\!]a \not\Downarrow_a$. Since the encoding $[\![\cdot]\!]$ is compositional on terms, there is a context D in π_{async} and a name $r \notin \text{fn}(P)$ such that $[\![C[P]a]\!] \equiv D[[\![P]\!]r]$, for all terms P, names a. Hence, $D[[\![M]\!]r] \Downarrow_a$ but $D[[\![N]\!]r] \not\Downarrow_a$. It follows that $[\![M]\!]r \not\approx_{\text{async}} [\![N]\!]r$. □

Unfortunately, the other direction of Proposition 6.4 seems to be much harder to prove. Proposition 6.4 requires that reductions in applied π can be simulated by reductions in asynchonous π, which is guaranteed by Proposition 6.3. The reverse direction would require that every possible asynchronous reduction sequence that starts and ends in an encoded applied term simulates a reduction seuquence in applied π. This appears credible, but a formal proof is still missing. We therefore can only conjecture that $[\![\cdot]\!]$ takes equivalences in applied π to equivalences in $\pi_a sync$.

Conjecture Let M, N be terms in applied π. Let $r \notin \text{fn}(M, N)$. If $M \approx N$ then $[\![M]\!]r \approx_{\text{async}} [\![N]\!]r$.

7 Conclusion

We have presented a modification of asynchronous π calculus that allows us to model sequential programming constructs in a simple way, using just macro expansions. We believe that this proposal might evolve into a formal foundation for programming languages that can express concurrent execution of processes but at the same time retain their sequential programming heritage. However, more work needs to be done until this goal is achieved.

In particular, we would like to get process equivalence criteria that are more tractable than the barbed congruence we have used. Another open question concerns the relationship between the process equivalence theory of applied π and the corresponding theory of the pure asynchronous calculus. Finally, it should be possible to define a typed version of applied π by generalizing Milner's sorting approach for π calculus [14].

Acknowledgments I'd like to thank John Maraist, for reading and commenting on previous drafts of this work, and Benjamin Pierce, for his thorough review, which was a great help in improving the paper.

References

[1] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] Dave Berry, Robin Milner, and David N. Turner. A semantics for ML concurrency primitives. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–129, January 1992.

[3] Gérard Boudol. Towards a lambda-calculus for concurrent and communicating systems. In J. Díaz and F. Orejas, editors, *Proceedings TAPSOFT '1989*, pages 149–161, New York, March 1989. Springer-Verlag. Lecture Notes in Computer Science 351.

[4] Gérard Boudol. Asynchrony and the pi-calculus. Research Report 1702, INRIA, May 1992.

[5] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8):15–42, August 1992.

[6] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[7] Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[9] Keiho Honda and Nobuko Yoshida. On reduction-based process semantics. In *Proc. 13th Conf. on Foundations of Softawre Technology and Theoretical Computer Science*, pages 373–387, December 1993.

[10] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proc. 5th European Conference on Object-Oriented Programming*, pages 133–147, July 1991. Springer LNCS 512.

[11] C.B. Jones. Process-algebraic foundations for an object-based design notation. Technical Report UMCS-93-10-1, University of Manchester, 1993.

[12] Ian Mason and Carolyn Talcott. Equivalence in functional languages with side effects. *Journal of Functional Programming*, 1(3):287–327, July 1991.

[13] Robin Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[14] Robin Milner. The polyadic π-calculus: A tutorial. Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Edinburgh University, October 1991.

[15] Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[16] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I + II. *Information and Computation*, 100:1–77, 1992.

[17] Robin Milner and D. Sangiorgi. Barbed bisimulation. In *Automata, Languages, and Programming, 19th International Colloquium*, 1992. Lecture Notes in Computer Science 623.

[18] Martin Odersky. A functional theory of local names. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 48–59, January 1994.

[19] Martin Odersky, Dan Rabin, and Paul Hudak. Call-by-name, assignment, and the lambda calculus. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 43–56, January 1993.

[20] United States Department of Defense. *The Programming Language Ada Reference Manual.* Springer-Verlag, 1980.

[21] Benjamin C. Pierce, Didier Rémy, and David N. Turner. A typed higher-order programming language based on the Pi-calculus. Draft report; available in the PICT distribution, July 1993.

[22] Andrew Pitts and Ian Stark. On the observable properties of higher order functions that dynamically create local names. In *SIPL '93 ACM SIGPLAN Workshop on State in Programming Languages, Copenhagen, Denmark*, pages 31–45, June 1993. Yale University Research Report YALEU/DCS/RR-968.

[23] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 293–305, June 1991.

[24] Davide Sangiorgi. An investigation into functions as processes. In *Proc. 9th International Conference on the Mathematical Foundation of Programming Semantics, New Orleans, Lousiana*, pages 143–159, April 1993.

[25] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[26] Vipin Swarup, Uday S. Reddy, and Evan Ireland. Assignments for applicative languages. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, pages 192–214. Springer-Verlag, August 1991. Lecture Notes in Computer Science 523.

[27] David Walker. π-calculus semantics of object-oriented programming languages. In Takayasu Ito and Albert R. Meyer, editors, *Proc. Theoretical Aspects of Computer Software*, pages 532–547. Springer-Verlag, September 1991. LNCS 526.

[28] Niklaus Wirth. *Programming in Modula-2.* Springer Verlag, 2nd edition, 1983.